Design, implementation and deployment of a cloud infrastructure at a large organization

Daniel Lobato García - Tutor: Jesus Carretero Pérez

Escuela Politécnica Superior

Universidad Carlos III de Madrid

A thesis submitted for the degree of

Bachelor of Science in Computer Science

2013

Dedicated to the Agile Infrastructure team at CERN, and everyone else who have supported me during the past year.

Abstract

Data centers, the giants who are powering the technology that drive the most significant businesses and research efforts, have undergone many changes throughout the past recent years. Modern widely-used tools, virtualization mechanisms, clouds, and fabric management are being implemented in order to reduce operational effort and allow engineers to offer Infrastructure as a Service with greater facility than ever. This document reviews the current state of the art and strategies when building a cloud. In addition to that, there is a big focus on the implementation and deployment of the chosen solution. Personal contributions to the ecosystem and an explanation of how they are relevant are also reviewed. Ideas and restrictions to be thought of in the context of the European Union are included as an appendix.

Contents

List of Figures

Chapter 1

Motivation

1.1 Introduction

CERN's data center is migrating its infrastructure from old provisioning (Quattor) and cloud computing (LxCloud) in-house tools that powered the data center during the early stages of the LHC experiment. As the amount of data grows, more and more people all around the world get involved with the project, and maintenance of these tools becomes a huge burden that might not be needed. This and the fact that there will be a remotely managed data center in Hungary that will comprise the tier-0 alongside Geneva's one forced the architecture board to choose other tools that are more well known industry wise and provide more benefits. The consensus was to migrate to tools, OpenStack for computing and Puppet for provisioning, using Foreman for managing the data center. These are young open source projects, but the community is vibrant and growing rapidly. NASA, Red Hat, Intel, and other companies are contributing as well as CERN to the development of these tools to be available for free to any scientific computing data center in the world.

As of now, the project generates 37 TB of raw data per day, of which around only 10 TB are actually useful. This makes the LHC experiment the most computing expensive project ever made, and tweaks to the way the data is processed through Hadoop, Hive, and other tools are made at CERN. There are 10Gbit/s switches that send this data to all of the Tier-1 centers to further analyze this data, and there are various bottlenecks both on the computing side and on the networking side. The new center in Hungary aims to alleviate the problem.

1.2 Thesis overview

One of the objectives of the objective of this thesis is to explain my contributions to a successful case. In addition to that, I will give a view of the current state of the art when it comes to building a scientific computing data infrastructure, explaining in detail how machines are automatically configured, how clusters of hosts are remote power controlled and other critical aspects of infrastructure. Other successful and failed business cases will be explained to let the reader choose their own range of

tools.

An addendum with law regulations concerning the hardware and the workers and a prospective budget are included in order to properly estimate new budgets for similar projects.

All in all, this document will give the reader a good insight on how to manage, build and deploy a modern cloud infrastructure.

1.3 Value propositions

In order to describe the values a cloud can provide to an organization, a definition of cloud should be made. Luckily, the National Institute of Standards and Technology (from here on NIST) has established a set of characteristics of cloud computing. These are considered the standard value proposition of the technology, which can be complemented by supplementary services provided by the vendor. These are:

1.3.1 Rapid elasticity

The cloud shall be seen as an infinite resource from the point of view of the consumers. Measures will be taken from the service provider to ensure smooth scaling.

1.3.2 Measured service

The service provider shall measure, monitor and control all relevant interactions of the consumer with the cloud. This allows the provider to plan growth accordingly, improve security and protect his liability among other benefits.

1.3.3 On-demand self-service

Consumers shall be able to get and release resources without any kind of human interaction. This saves manpower that can be better spent on improving the service.

1.3.4 Broad network access

The cloud shall be accessible through any kind of client, without major geographical restrictions, and access will be regulated by standard protocols.

1.3.5 Resource pooling

Cloud providers shall serve their resources in a multi-tenant model. That is, a single consumer or several consumers can be served by a tenant of their choice. Resources can include memory, storage, computing power and network bandwidth.

A considerable amount of savings on staffed support needs to be considered as well. As much as these propositions benefit the users, providers need to analyze the

trade-offs that they might incur in, for instance not being able to find professionals to deploy it.

Chapter 2

State of the art

Cloud computing ideas can be traced back to the 60s. J.C.R Licklider (ARPANET) himself used to refer to an 'Intergalactic Computer Network' that would provide compute resources on demand through a global network. All in all, cloud computing main ideas are not too different from Project MAC's. Amazon EC2, Salesforce, and a handful of big players have joined in in the decade of the 00s to offer computing power to any user in the way we today call cloud computing.

2.1 Standardization efforts

In addition to the guidelines and definition of cloud computing established by the NIST -mentioned in Chapter 1-, a few other groups have emerged. These standards aim to regulate how deployment, provisioning and quota management/monitoring. These groups are the Cloud Standards Customer Council (IBM), European Telecommunication Standards Institute and the Open Cloud Consortium (University of Chicago). There are many other groups trying to find standards for the characteristics mentioned in this chapter, but as of now, the most respected group is undoubtedly the NIST.

All in all, these groups were created to provide a standard framework for new clouds, so the adoption of best practices and other recommendations is still not very well-known nor extended. This document will take them into account, filtering out parts of these standards that may have become obsolete.

2.2 API interoperability

There are four defined cases as per 'The Role of Standards in Cloud Computing Interoperability' (CMU). These are Workload Migration, Data Migration, User Authentication, and Workload Management.

Even though all major vendors (VMWare, Amazon, Openstack) offer APIs for their services, they are vastly different. Most of these REST or SOAP APIs do not allow to transfer workload to other clouds. There are open standards that allow

vendors use solutions from their peers, such as Virtual Hard Drive (VHD), or Amazon Machine Image (AMI).

User authentication is clearly another feature that all clouds share, and an interoperable API is not far from being reached. AWS IAM, OAuth, OpenID and other authentication systems are the de-facto standards used in most modern cloud systems.

Cloud Data Management Interface (CDMI), a body that sets standards on how to handle storage in a standard manner, is in charge of data and storage management interfaces, but unfortunately both the standards and the products using are lined-up for the future.

There is barely any work done on the deployment of virtualized machines, and all vendors decide to implement their own API. Efforts such as the Fog project for the Ruby language, or libvirt for C try to provide bindings for all clouds until there is an agreement on this issue.

2.3 Security

Security in clouds is not only a hot topic in research but a controversial one. Since there are security issues that can arise from many fronts, in this paper only issues whose cause is in the definition of clouds by NIST. Any other issues might not be particular to clouds and might just be details of the implementation.

- Virtual machine escape
  - One of the most important security issues, largely unexploited on AWS and Openstack. There are penetration tools which exploit this vulnerability on VMWare, mostly because AWS hypervisors are hard to reach, and Openstack is young. VMWare offers specialized support, with no guarantees, for hypervisors to avoid this.
- Session hijacking
  - Shared VMs and operations through the API can exploit session hijacking. There have been problems on this, and since HTTP is a stateless protocol, a need for sessions on the hypervisor is needed. All vendors provide SSL compatible APIs. Protection against local malicious users (within the same LAN as the hypervisor/vm) requires a comprehensive network securitization. Applications like Kochure help audit possible hijackers.
- Data recovery
  - The cloud characteristics of pooling and elasticity can severely damage the ability of a system to recover compromised data. Similarly, if not handled properly, other users could try to read or write data on other virtual machines through the hypervisor.

In addition to the aforementioned vulnerabilities, it has become a common practice for crackers to create virtual machines containing rootkits and trojans. After these machines are distributed over the Internet, the crackers have access to every machine with this rootkit. These virtual machines, or even just programs, often are bundled with versions of the original product so that the victims never notice they are being spied.

2.4 Quota management

Quota management, also referred to as cloud management, is the set of challenges that a cloud infrastructure brings in terms of auto scaling, billing, tracking of resources, and others.

Auto scaling in the cloud saves both time and human power. Summing up, auto scaling is how to set up your cloud to provide and provision new resources on end-user demand. This usually involves the client setting up some predictive models based on algorithms and creating machines after them.

Some of these algorithms include the following techniques:

- Response time analysis
- Static threshold-based rules
- Reinforcement learning (Markov Decision Process and Q-Learning)

Reinforcement learning is indeed the bet most cloud services have made for the future. There are many techniques that can be used for this, but probably the two most widely used are the search for a criterion of optimality or value function approaches for multi-criteria decision analysis. Algorithms used for reinforcement learning find out which decisions are good mostly based on experience.

A reinforcement learning agent in a cloud needs to observe what are the effects of the quota policies, find out how to waste fewer resources and other considerations (reward) and optimize for a situation where the reward is maximum. Since the rules to compute the reward are stochastic, reinforcement learning is well suited for problems where a long-term maximum reward is better than short-term best effort rewards.

2.4.1 Criterion of optimality

Given a problem, criterion of optimality is a technique that comes in handy when there is a clear penalty or reward for making certain decisions. This technique assumes the agent knows what could be the maximum expected return.

2.4.2 Value function approaches

Value function approaches focuses on keeping an optimal state for just one policy, instead of choosing which is the best among a pool of policies defined by a human. The state can be defined for instance, in terms of a combination of memory and

CPU available at the hypervisor, for a group of users, or anything defined in terms of reward and penalty by the cloud maintainer.

When started, the agent will identify a set of states that could for instance be defined in terms of typical actions in a cloud setting, deleting a virtual machine, provisioning it, etcetera. Once there are enough data, it will start making corrections to the policy until it finds an optimal state.

Unlike in other type of problems, cloud operations are limited in number, therefore value function approaches allows vendors to utilize Monte Carlo simulations to provide clients with a number of optimal policies so that they do not normally have to be tweaked by the clients. Amazon uses this method on their quota systems, but it allow users to change their quota algorithms shall they decide to use custom tenants.

2.5 Batch management

Batch control and scheduling are not strictly NIST requirements of cloud computing. Nonetheless, most vendors are offering batch management to their users. Because of this, the topic will be covered to some extent.

Cloud Foundry offers an API to submit batch processing jobs to a RabbitMQ queue. Virtual machines are bootstrapped as needed and it does not offer any parallelization. As parallelization is a major requisite in most batch environments, solutions that offer easy parallelization of jobs and auto launching of new workers are preferred.

This is why in the field of high energy physics, where data analysis is usually easy to parallelize, two of the top solutions can be found. The University of Victoria in British Columbia, Canada, has developed Cloud Scheduler, which provides an interface to submit jobs and lets the system decide on how much computing capacity is needed to automatically bootstrap virtual machines that take on these jobs. This is a major step ahead of Cloud Foundry which does not offer such an option. Most non-vendor specific solutions can connect to an API similar to Amazon's EC2, which is the standard de facto. Luckily, the most developed open source solution to build cloud, Openstack, offers a socket for this.

Chapter 3

Use cases and industry examples

Clouds are being heavily used on a variety of environments for several purposes. On this chapter, some of the most prominent use cases will be explained. Some of the most important industry examples will be explained as well to provide models that could be followed.

3.1 Scientific computing

Scientific computing has for long taken the lead when it came to computational demands. Nonetheless, these days Internet companies demands have grown to a scale bigger than that of most research. Still, scientific computing is one of the most demanding use cases for a cloud.

Since scientific experiments needs can be wildly different, three different major sub use cases can be studied separately.

3.1.1 High performance computing (HPC)

High performance computing focuses on *capability computing*. Capability computing is the force driving most HPC systems these days, which is solving the biggest problems in the smallest amount of time. This is substantially different from *capacity computing*, which demands the system to run as many problems as possible in a cost-effective way.

Supercomputers built for scientific research do not always benefit from a cloud approach to managing resources, especially when the hardware is specifically built to solve certain kind of problems. To put it simply, current HPC supercomputers let a researcher run a problem on 700 cores at the same time for a limited amount of time, which is something no vendor offers as of now since HPC is not a key market. Amazon has started a limited program which allows US universities to rent clusters from its 17,000 cores cluster and 10 Gb Ethernet connection, in chunks of clusters containing 64 cores each. This is not enough especially when interconnection needs are high.

There are trends in HPC which incorporate volunteer computers to solve large-scale *embarrassingly parallel* problems such as BOINC or Folding@Home (protein folding), but most traditional simulations supercomputers were built for, like dynamic fluids simulations are still unfeasible with any of the approaches mentioned above. All in all, it can be concluded that supercomputers are still irreplaceable in a lot of situations when it comes to scientific computing, and the state of the art for clouds does not offer any alternative.

3.1.2 High-throughput computing (HTC)

HTC in contrast is more concerned with long-term jobs instead of delivering the maximum amount of operations per second. In order to do so, jobs are spanned across as many resources as possible and then put back together. Nodes processing these micro-tasks need to be extremely reliable as the ability of the system to produce a result is usually dependent on joining the results of all micro-tasks.

HTCondor and PBS are the leading edge programs when it comes to distribute the incoming jobs into the available computing resources.

Clouds are meant to be resilient after the failure of one of its virtualized nodes, so this is a model that definitely fits in this approach to distributed computing resources. Unfortunately, there is not any software available or any vendor that commercializes this, most likely because of how small HTC is relative to the market of selling computing resources.

3.1.3 Industry examples

3.1.3.1 CERN

CERN's system migrated from a homebrew stack of tools, which they developed alongside other cutting-edge research labs in the early 2000s. Lately, computational needs for commercial companies have outgrown those of some research labs. As a result of that, CERN has decided to take advantage of the existent tools to deploy a toolchain based on Openstack for virtualization and networking and Puppet for configuration management. This allows the lab to iterate on the tools since they are open source, and contribute back with their own customizations.

3.1.3.2 Genome Quebec - McGill - University of Waterloo

A paper co-written in 2009 describes the experiences of a group formed by people of these three research facilities with Hadoop data processing in a cloud. In order to get a better understanding of the whole situation, they put their sights on Google's web processing stack. Google's stack (MapReduce, Google File System, Big Table) is very tailored to specific applications. To simulate this without proprietary tools they chose to use Hadoop's MapReduce, Distributed File System, and HBase for sparse structured data. Processing of sequences of microscope images of live cells was possible by letting clients submit their inputs to MapReduce, and the cloud would take care of the rest. It required minimal customization of Hadoop's MapReduce,

and they contributed back to the community with a module that allows whole DB tables or directories as input data for the jobs, while the original MapReduce only lets the user upload single structured files.

3.2 Cloud Service models

In addition to the constraints mentioned in the motivation chapter (resource pooling, elasticity, on-demand self service, etc...), clouds are recognized to be able to serve their business in three different ways. Each of the service models fill a business need and is geared for a certain kind of user.

3.2.1 Software as a service (SaaS)

Users are able to benefit from software, without having it installed in their machines. Information is served from the Internet, and its business model is normally monetization through subscription. Usually applications deployed in this model cater to the general crowd, and since costs per user are low in this environment, it is common to offer a free version of the product. Main advantages are that operational costs are lower than in a self-managed environment, but user data privacy can be compromised as administrations do not have access to the system in its entirety.

3.2.2 Platform as a service (PaaS)

PaaS services offer a set of tools, usually a programming language execution environment, and similar utilities so that users can deploy their applications on this setup. Usual complaints from users ask for more control to debug their applications, and more flexibility, but usually that compromise can only be reached switching to having own servers or using infrastructure as a service servers.

3.2.3 Infrastructure as a service (IaaS)

IaaS offers real compute resources, usually in the form of virtual machines, network load balancers, storage servers, and similar appliances, under the NIST cloud premises. This gives the maximum amount of flexibility to users as they have access to the real machines where their code is deployed, and they can tune it to their needs. Privacy is still a concern as reversed virtual machine escape could happen and administrators might have access to confidential data. These machines normally run under a hypervisor that is able to scale up or down the resources depending on the needs of the client. The cloud described in this document will be providing IaaS. The reason being users can easily build their own PaaS and SaaS businesses on top of our cloud, while the opposite is not true as users of a PaaS service do not have much of a choice over the backend supporting their tools.

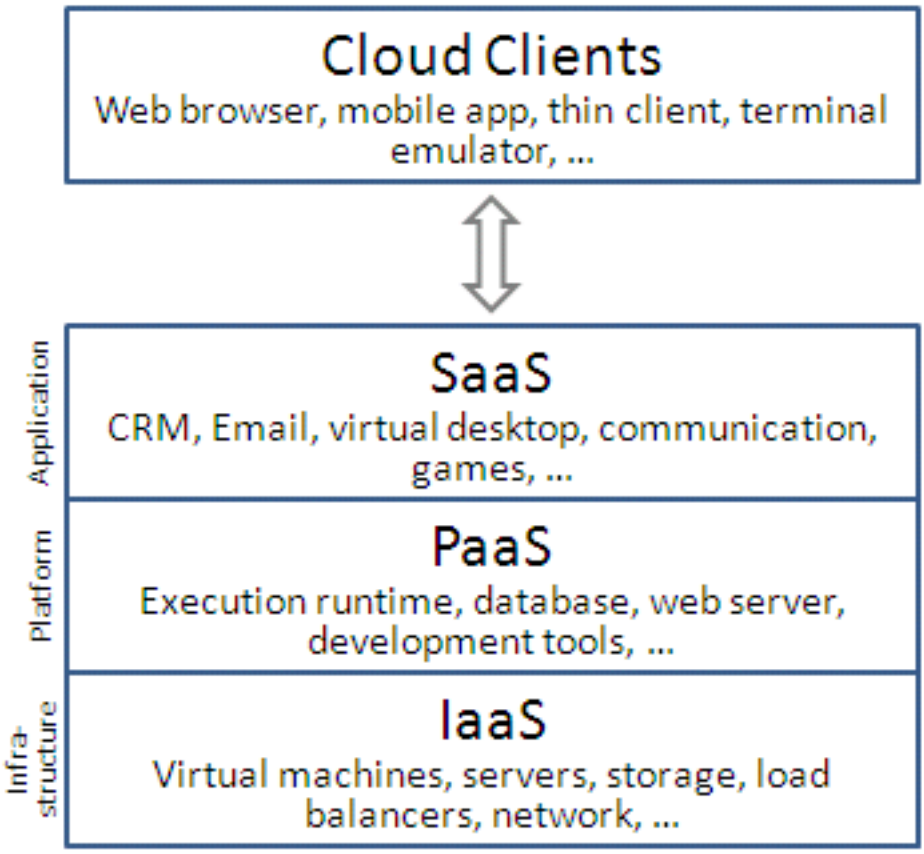

Figure 3.1: Cloud stack

Chapter 4

Design and Tools

The stack of tools chosen for a cloud is easily the most deciding factor on whether it will be a success or not. In recent years, a number of open source solutions have arisen. As of now there is no need for proprietary solutions in any of the parts of a cloud, this would be a disadvantage if it needed to be integrated with other services at a deployment. Taking into account not only the price but the extensibility, maintainability and features of current solutions, only a brief overview will be given about proprietary solutions since they hardly offer an edge on anything anymore.

4.1 Stack definition

A basic cloud stack needs to include tools to:

- Virtualize, distribute and schedule processes on the virtualized clusters
- Define and provision configurations
- Allow programmatic remote execution on clusters

These points define the domain of work. Additional needs, such as monitoring, remote power control, and others are conveniences and details of every particular implementation, which can be changed easily. It is very unlikely that the choices to fulfill the needs mentioned above will ever change, and if they do, it can even be considered as a new cloud.

After learning about the experiences with several cloud providers, it came clear that *Amazon Web Services (AWS)* are a step ahead the competition. Nonetheless, AWS systems are largely proprietary and research on how they work internally is restricted to what they present on conferences.

Openstack is catching up on most features AWS has, other than autoscaling, and offers some interesting components that AWS does not. Unlike AWS, Openstack is mostly offered as a pack of software modules to implement on your physical hardware. This makes it the perfect choice for learning how to set up a cloud, which are the main parts, and what role do they play. For instance, for small installations, a component that allows you to make network topologies might not be needed.

Aside from virtualization, a system that provisions configurations to nodes according to definitions written by the users will be needed. As of now, *Puppet* and *Chef* are the most modern tools, that let users customize their node definitions using Ruby. Software like *Quattor* or *CFEngine*, for long leaders of IT configuration management is coming to an end due to its difficulty to extend node definitions arbitrarily and complicated setups for large organizations.

Puppet will be the choice. Its proven scalability, enormous community and extensibility make it a perfect candidate to understand how a configuration management system should work. A Domain Specific Language (DSL) is used to provide node definitions, which is easier to teach to new users than a whole new language (Ruby), which is the standard way of providing node definitions in Chef.

The fact all the chosen tools are open source is actually key to the future of this infrastructure. Having the ability to include custom modifications without any approval of a vendor, and being able to debug the software line by line will likely help scale the infrastructure and implement any features that are not already there.

4.2 Virtualization

Openstack Nova is the component that will be the main fabric controller. The architecture of this component is also separated into many blocks concerning each of the most important parts of a virtualization suite.

The following diagram shows its inner interconnections very well:

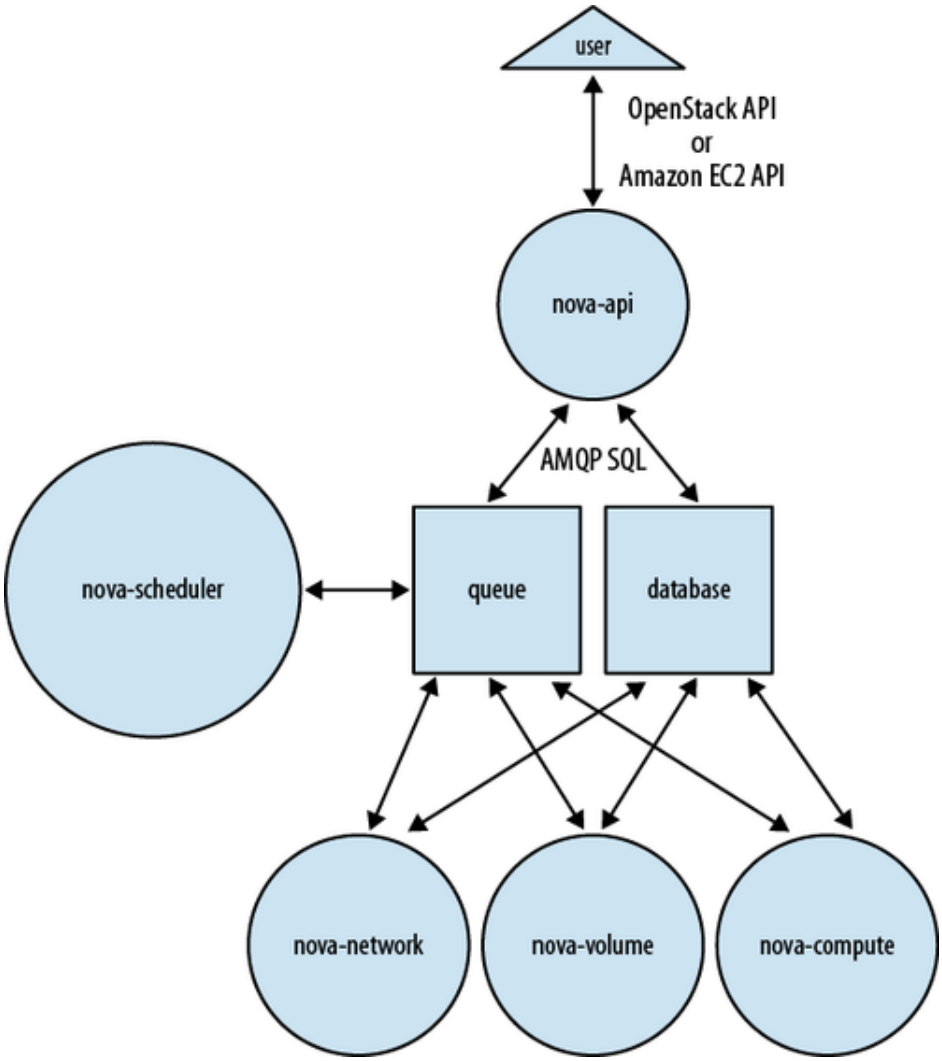

Figure 4.1: Nova virtualization suite

Nova-api is the core of the virtualization fabric controller. The user interfaces directly with it through the command line using API calls through HTTP, or can use another component of Openstack (Horizon, the dashboard), to submit requests through a web UI. It is the only interface the clients will interact with to virtualize servers.

Nova-scheduler is constantly taking virtual machine creation requests from the queue, finding the less busy host, and creating the virtual machine on the aforementioned host. In order to do so, it offers three scheduling algorithms, but if the user needs something more elaborate, a custom algorithm written by any hypervisor manager can be plugged in. The three default schedulers include Simple (find less loaded host), Chance (choose random host), Zone (choose random host within availability zone).

Compute essentially reads calls made from the API or nova-scheduler and runs them

according to whichever virtual machine model is needed. In practice, this means nova-compute needs to 'know' several APIs to interface with other virtualization programs. Libvirt, Windows Management Instrumentation, vSphere, and others, are currently supported thus allowing virtual machines of various kinds to be created. In a large organization, this could mean KVM for Unix virtual machines, keeping a large pool of Linux hypervisors (all kernels are compatible with KVM) and allowing to virtualize Windows guests if needed.

Nova-volume is a block storage service that provides persistent volumes to virtual machines, similar to what Amazon Elastic Block Storage offers. This mainly allows users to have volumes that can be mounted on several compute machines. IOPS depend on the kind of volume selected, but no overhead is added, so there is no penalty for the user on using this versus mounting the volume directly on the virtual machine. This topic will be revisited on section 4.9, long term data storage.

Nova-network a networking component will be revisited as well on 4.3 because this will be substituted by another component that allows virtual machines to work on user-defined network topologies.

4.2.1 High-availability scheduling queue and database

The queue and database in Openstack are critical to the scalability of any deployment. All Openstack components are stateless, so it is necessary that these components are highly available, fast and highly scalable.

Summing up, failures on the queue will force the system to freeze and not respond to user changes. If some events, for instance the removal of a VM were in course during the failure on the message queue, it is likely they will be on a disrupted state and resources are wasted.

As for the database, that would have a serious effect for every user as VMs will not be able to tell who should have access to them, which network are they in, and similar issues. Remember that the database will keep the *topology* of our deployment

Openstack uses SQLAlchemy, a well-known Python library that acts as an Object Relational Mapper between the SQL database itself and the programmer. This means Openstack would normally support any database also supported by SQLAlchemy. Since the default database is MySQL which has proven sufficient even for large deployments, focus will be on the different MySQL setups that will lead to a high-availability system. See Highly Available MySQL, on chapter Deployments for more information on the different alternatives for a highly available MySQL system.

On the other hand, only RabbitMQ is supported. Asynchronous message queues are normally not highly available even with durable (messages written in disk after RAM is full) queues. Nonetheless, we can provide a best-effort message queue with some trade offs. This topic will be further explained on chapter Deployments, Highly Available RabbitMQ.

4.2.2 Images service

Our image registry, Openstack Glance will basically support two web services. An API for CRUD image operations and an API for CRUD image metadata. The reason for this split is because of the different nature of work the result of operations run on large versus small files require. The images service will be a key part of the infrastructure, interfacing with users when they upload their own images and associated kickstart templates, and likewise our virtualization suite (Nova) will interact with it to provision the VMs with a basic setup.

Choosing a sensible backend service is possibly the most important part of it. Block storage is also provided by Openstack in the form of the Swift product, but a distributed file system, a S3-compatible system, or regular HTTP could be used too. Most likely unless you are building your own images, reading files from HTTP is a sure way to go if there is enough trust on your images source. Even if the answer to that is negative, it might be worth it to setup your own highly available HTTP server cluster and use it as a source for Glance, which is an easy task to do. Most likely even in large deployments, sensible DNS load balancing along with several HTTP backends will allow to scale horizontally very well.

4.3 Networking

4.3.1 VLANs and Floating IPs

Virtual, physical networking in Openstack relies on internal *Virtual LAN*s (VLAN). This is motivated by a desire to keep different virtualization tenants unable to reach each other directly. This allows users to create their instances in separate *virtual availability zones* that help them offer high availability services. Effectively these machines are running under the same physical node, physical network interface, but they are unaware of each other and the only bridge between them is a switch. This is why this model is often called "*Layer 2 isolation*" because the instances are isolated at the data link level. A more advanced example follows in this figure:

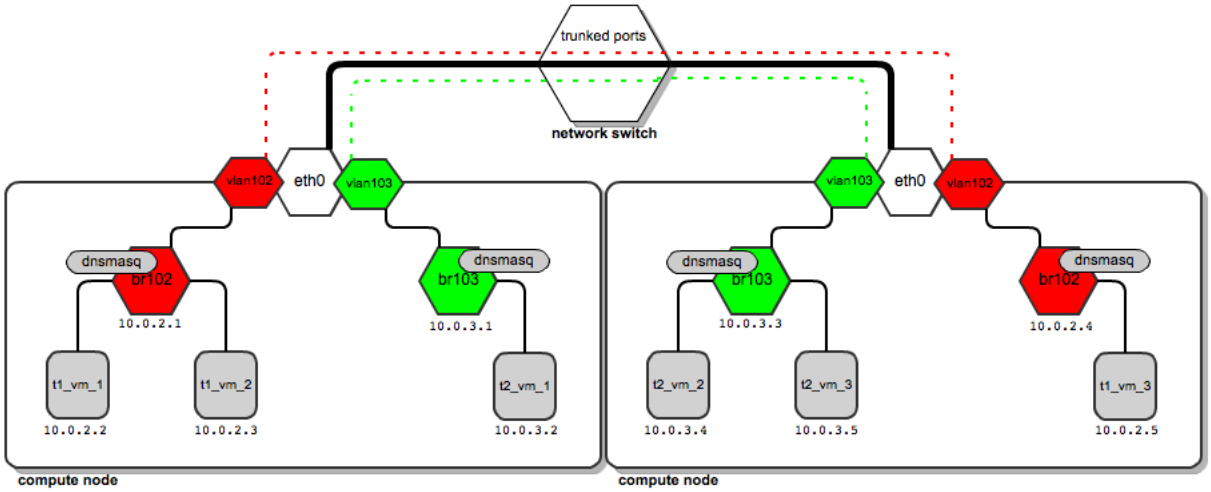

Figure 4.2: Network with two VLANs under two separate physical nodes

The two white boxes represent different nodes. Two different nodes have a different network interface. Nonetheless, *virtual availability zones* need to overcome being on several machines. It would be inconvenient for a user to be limited to deploy a virtual availability zone in one node.

In the example above, there is a red and a green virtual availability zones. *VlanManager*, a component of Openstack, makes this scenario possible, where all red virtual nodes (and virtual interfaces) are on the same virtual network zone, even if they are across different machines. This technology also allows users to create sets of *floating IPs*, reachable from wherever they want, inside or outside the data center. All of the functionalities explained above are available on Openstack Nova at the moment of writing.

4.3.2 General Network Topology

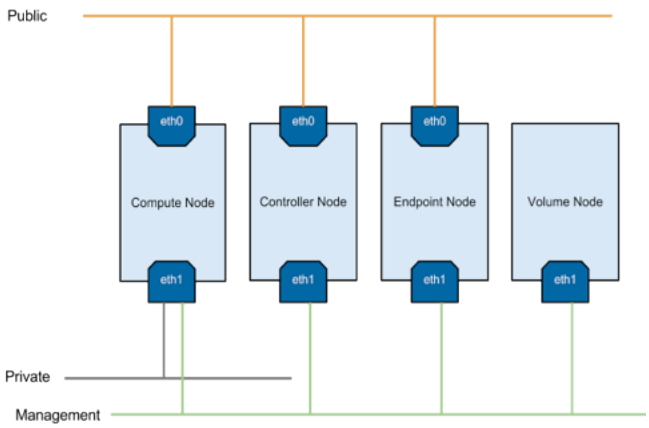

Figure 4.3: Network

The network should be separated into three different subnetworks to provide enhanced security:

- Public network: Connects different virtual IPs with requests coming in from the internet outside the deployment. Exposes nodes to the internet.
- Private network: Bridges all compute nodes, using *VLanManager* as explained above, it gets split into different VLANs.
- Management network: As per virtualization section of this chapter, this network exchanges information between all the Openstack components. Its setup consists on fixed IPs created by a NAT connecting nodes, firewalled from the rest of the world.

4.3.3 Load balancing as a service

Load balancers are a key piece of software that distribute requests between application servers. This feature is of utmost importance to almost all production services as it allows having a single endpoint to which clients issue requests, this is easier for the user than having them point to the different backend servers. In addition to that, ideally the load balancer will not assign too much load to any servers behind so that if a lot of traffic comes in and if any of the backend nodes breaks down, the production service will not go down as other nodes will handle its requests.

There are *physical* and *software* load balancers, and the focus of this subsection will be on how to provide load balancing to users of your cloud. Physical load balancing is a topic that has to do more with physical networking and less with virtualization, and as such, it will not be covered.

Since the applications clients of the cloud will differ, some of them will require high throughput, some others might need to keep connections alive between clients

and hosts... all in all, not a single strategy for load balancing will fit all use cases and it is best to study all possibilities.

4.3.3.1 Layer 4 load balancing - Transport level

Layer 4 load balancing is named after the OSI layer model because requests are only inspected at the transport level, usually TCP or UDP. As such, it is faster than checking layer 7 headers. A round robin DNS policy is the most common load balancing algorithm in layer 4.

4.3.3.2 Layer 7 load balancing - Application level

Layer 7 load balancing redirects requests to whichever application is used. This way requests asking for precompiled assets, static assets, etc... can be served in a more effective way.

Another benefit of inspecting the requests down to layer 7, is that backends can use *cookies* to tell the balancer that when a request with a certain cookie comes through, it should go to whichever balancer the cookie specifies.

There is a potential speed gain when HTTP load balancing is used. Using TCP keepalive, the user can establish a session with the balancer, and the balancer establish a session with a backend, and avoid performing the 3-way handshake every time a request comes through, as it would be the case with layer 4 load balancing. It is not the best practice to perform stateful operations in the balancer, but if the situation requires it, it can easily be done.

4.3.3.3 Available solutions

Given the stack of virtualization and networking is based on Openstack, it is a natural choice to consider using Openstack Neutron LbaaS (*load balancing as a service*) plugin.

This plugin will tie Neutron networking capabilities and models to actual devices. In practice, this means models such as *virtual ips* (hereupon 'vips') will have a layer on top of it to create load balancers and assign members to them. Other functionalities such as a firewall and VPNs gateways can also come down the line as the plugin matures.

What is described above can be accessed through Horizon (web dashboard) or a REST API. The latter allows clients to use configuration management to deploy their load balancers at configuration time. For instance, a node can automatically add itself to the load balancer after it has been configured. Another use case is in case of human error that might render a member unreachable by the load balancer the member can fix it without human intervention.

The following diagrams display how is a new virtual IP created and stored internally in Openstack using Nova and the LbaaS plugin for Neutron:

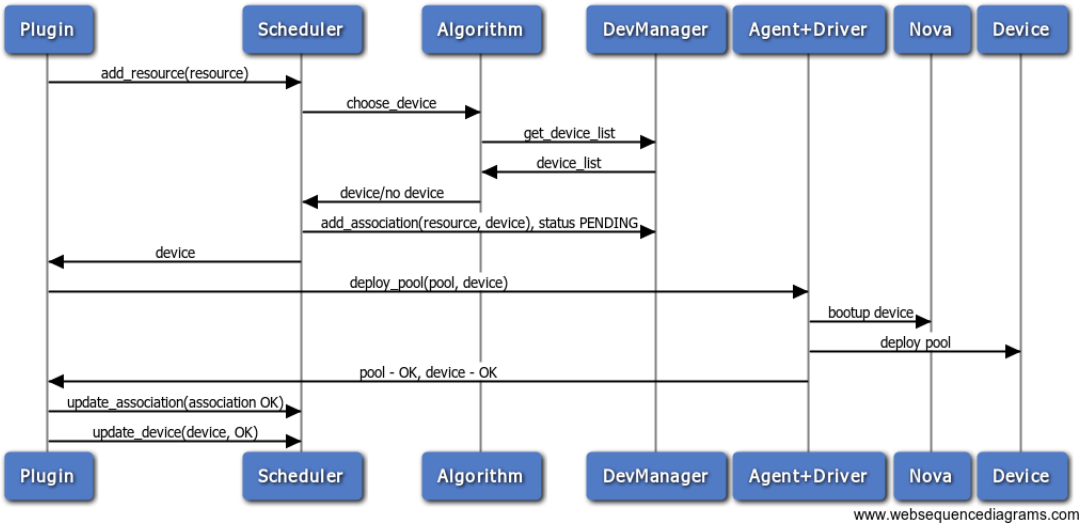

Figure 4.4: Virtual IP creation

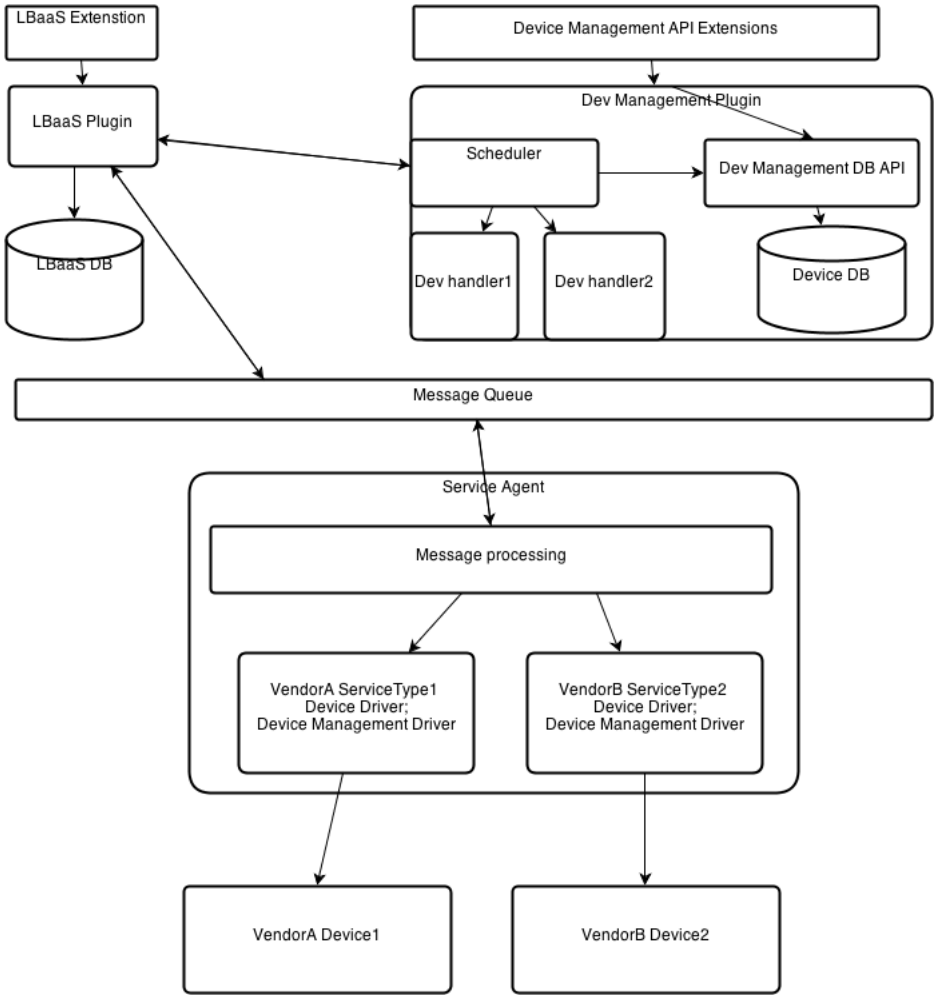

Figure 4.5: Neutron LbaaS architecture

An API definition for the following operations can be found here (https://wiki.openstack.org/wiki/N

- Create/Update/Delete VIPs
- CRUD Pools
- CRUD Pool members
- CRUD Health monitors

4.4 Configuration Management

Unfortunately, configuration management, as in automating configuration of thousands of machines, is an area that has not been well developed until recently in part due to cloud providers making servers cheaper than ever. Basically, a tool that allows any user, technical or not, to define the state of a machine, is what is needed. This way users can easily create servers of any kind on the cloud. Of course, some this whole configuration system will need itself a number of servers in order to serve properly to the needs of its users.

Luckily, it is possible to leverage the operational experience that built the two of the top contenders in the configuration management arena. *Chef* and *Puppet* are the two leaders in this space with two very different approaches to solve the problem. Chef is an *imperative* system which essentially means that the user has to figure out the operations that are run to turn the system into what is desired.

Puppet is a *declarative* system which means that the program can understand how to turn a system from its current status to whatever the user wants it to be. In such an environment where the users do not normally need to know about the internals of the cloud itself, it is likely that they will not know the operations needed to configure a machine but they will know how the machine will look like if it is, say, a Hadoop Node.

This choice will also help since users can run Puppet indefinitely as its operations are *idempotent*, so they can run them as many times as they wish without fear of breaking the hosts.

Puppet will run in a *master-client setup*, where the clients contact the master for its configuration, then a recipe is compiled and runs on the client. All in all, this setup will let the configuration management more or less scale horizontally.

Of course, details about the deployment of a truly easy to scale, performant Puppet master solution will follow on the next chapter, deployment.

4.4.1 Catalogs

Puppet will accomplish its goal by compiling a catalog of the changes required in the host. This catalog is created from the resources that the user specifies on the manifest. Internally, it is a graph of dependencies that follows the changes needed to be done to ensure changing to the current state of a machine to the desired one (specified on the manifest).

Package managers, and generally any program dealing with the configuration of a system are single threaded, simply because usually its not worth the effort to make the whole program multi-threaded when most of the time parts of the execution will be stopped to avoid race conditions and similar bugs. Similarly, Puppet also runs in a single thread, and the graph that represents the catalog is read as a list from top to bottom and changes are applied likewise.

4.4.2 Modules storage

As per section 4.4, Puppet will need the definitions of the configuration of these machines. These are *.pp files written in a DSL based off Ruby. When looking for a storage system for this, the Puppet masters themselves would work, but this would require a safe way for users to push their configurations (namely Puppet manifests) to the masters. Since such a tool is unrealistic, and any security problem will lead on compromising the Puppet masters themselves, it is best to find an alternative storage.

There should be some way of keeping track of what happened in the past, and how did these configuration definitions evolve over time. A version control system such as git is a good way of achieving both goals at the same time. For instance, a user might be interested on knowing what configurations changes made its server unusable while two days ago it was perfectly fine.

Puppet can easily read manifests from git repositories, and while setting up a git repository service is out of scope for this paper, a workflow compatible with a large organization will be provided in the next chapter, modules workflow.

4.5 Users perspective

Ideally, users would not need to use scripts unless they need to automate some process. In order to spin up virtual machines, manage the network availability zones, and so on, Openstack provides a tool called Horizon akin to the Amazon EC2 dashboard.

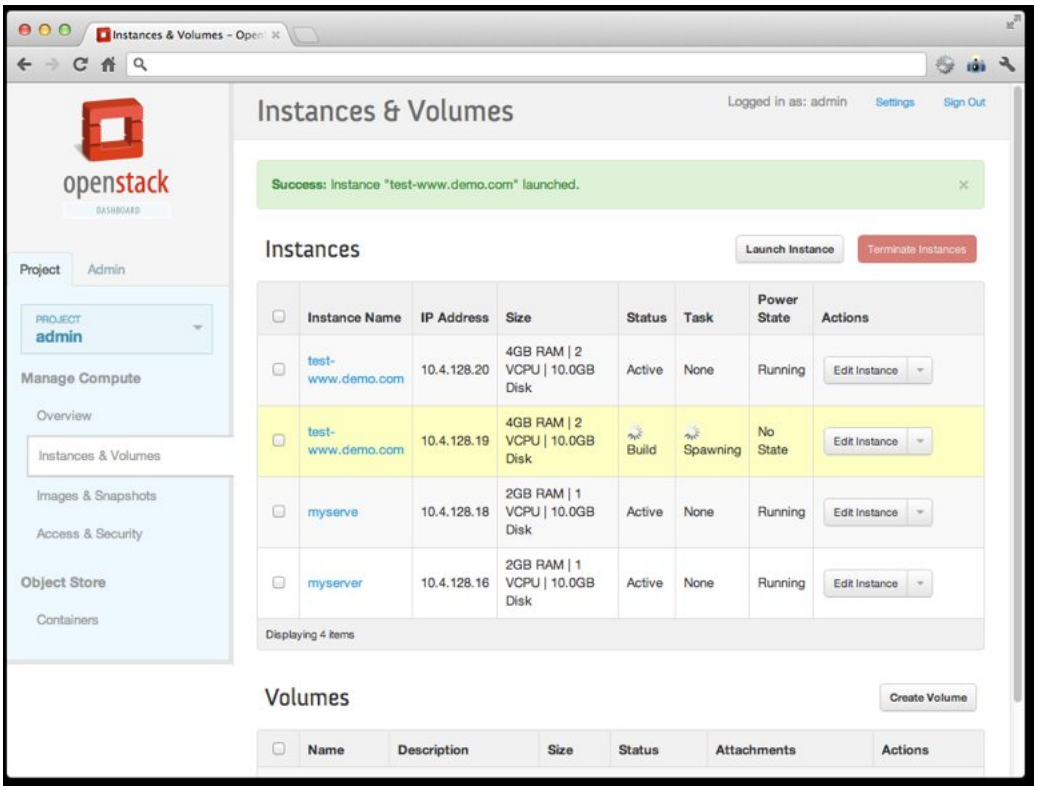

Figure 4.6: List of Instances in Horizon

Nonetheless, this tool is basic and provides the simplest possible service. Since our machines can be configured using Puppet as mentioned in previous sections, we can make this easy for our users.

The load balancing service can be integrated with Horizon seamlessly, and users can create their load balancers and pools of members through the UI as well as through the REST API explained in the previous section.

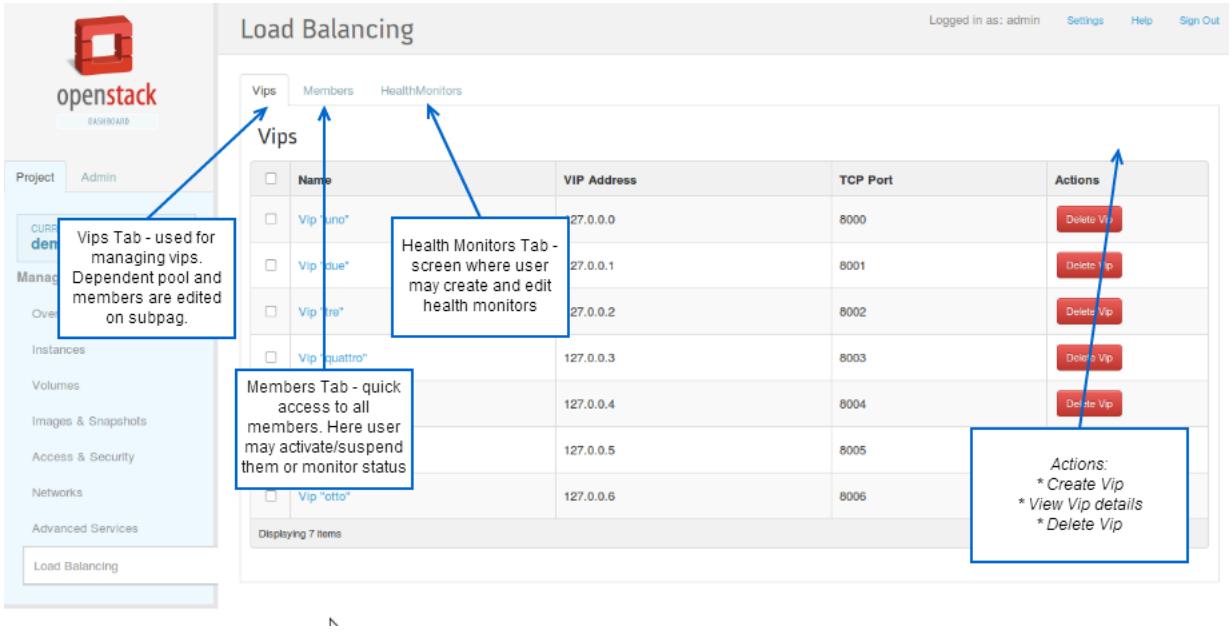

Figure 4.7: Neutron LbaaS exposed through Horizon

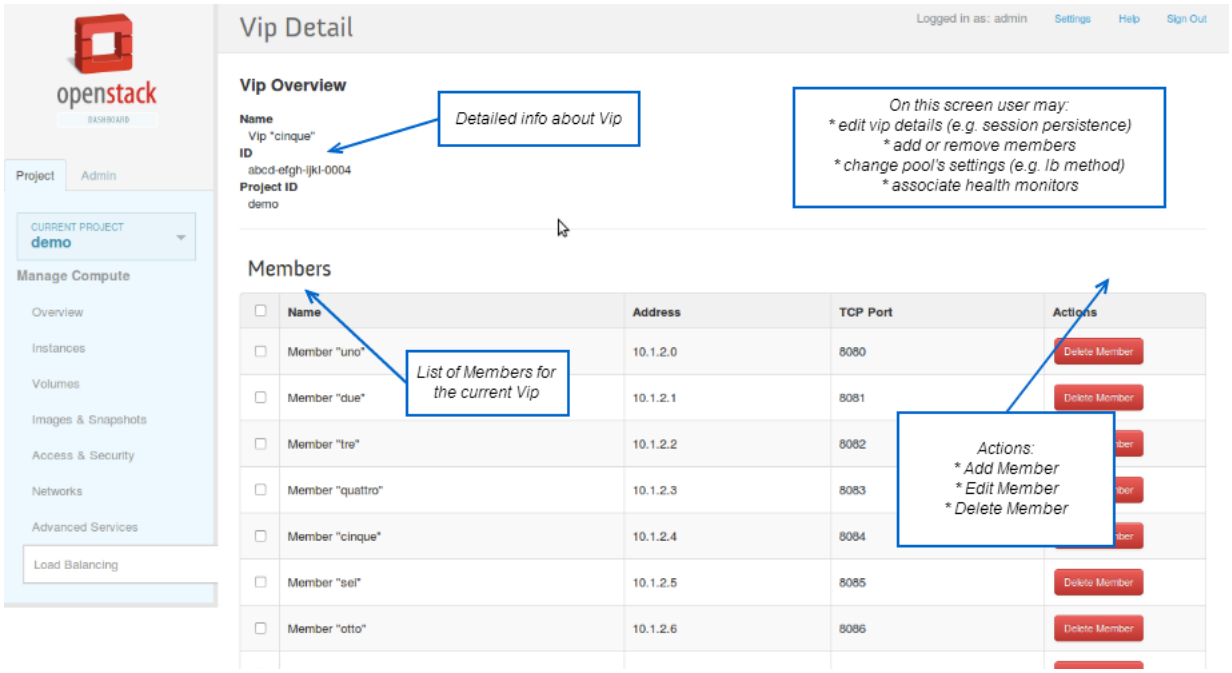

Figure 4.8: Details of Virtual IP in Neutron exposed through Horizon

4.5.1 External Node Classifier

An External Node Classifier is a program that given a host name, finds the classes it needs to be provisioned. In our deployment, this means that when a machine is created, it would be nice to let users choose which kind of machine they would want,

and boot it. In order to do so, we will group our machines in *host groups,* these being groups of machines with the same behavior, for instance, Apache servers.

Currently, there is only one tool that lets you do this in an integrated way with Openstack. Foreman, an open source project mainly backed by Red Hat is connected to the Puppet masters and can be used as a source of the desired configuration of a machine. In such an environment, if the user wanted to change what a machine is used for, something as simple as changing the host group will mean that the machine will get provisioned different Puppet classes and will be different.

Host groups manifests can be defined by the users to allow them to customize their virtual machines. For administrators, Foreman can also be a useful tool to perform power operations and read reports on the configuration attempts run on the machines.

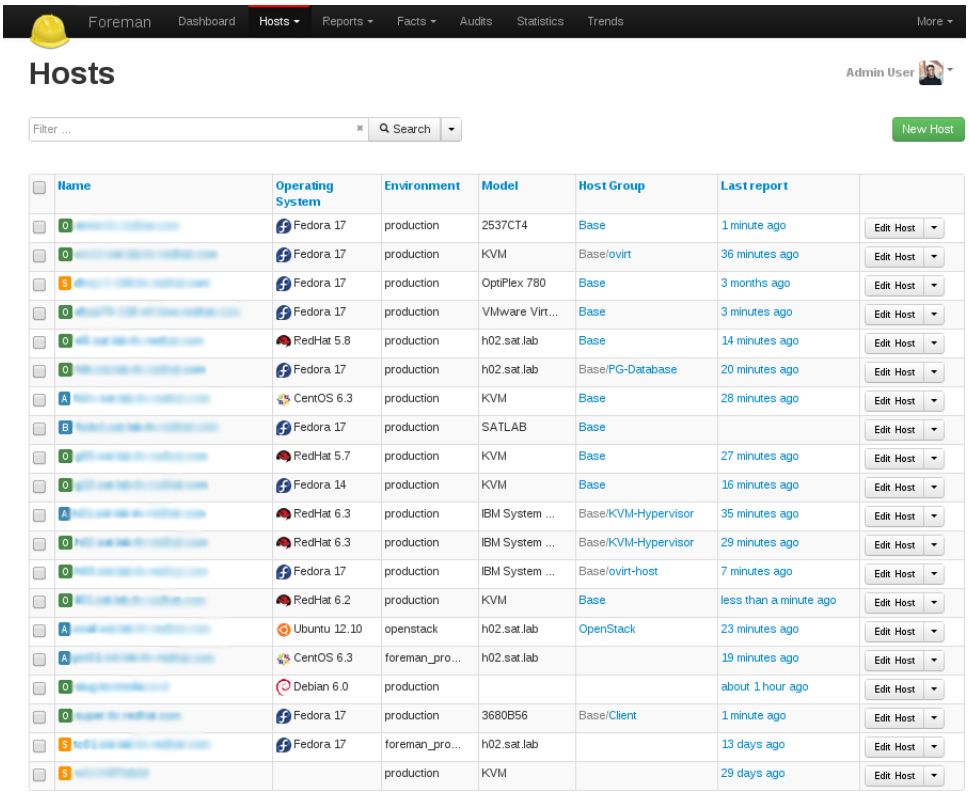

Figure 4.9: List of Foreman hosts

Details on the deployment of a highly available scalable Foreman follow on the next chapter.

4.6 Continuous integration

Given that you can have different environments in Puppet, it is a good idea to implement continuous integration on the production environment to make sure modules from other environments can be safely moved to production.

Because of the very nature of code that is meant to create an infrastructure, regular unit tests are not enough. The basic idea is to bring up a basic cluster of machines which will be configured with the Puppet modules. Tests will check the monitoring of this infrastructure, make sure that the cluster is working even when a node is not, etc..

After tests pass, VMs are set up on a NAT on the hypervisor, which is either the git repository server or a puppet master. This step tests whether the manifests do what it was expected of them, so it can be considered a unit test.

When the VMs have been set up, the test master waits until the configuration has been applied everywhere, be it just one machine or a whole cluster. This step is preparation for the functional tests.

Finally, the final tests submit some requests to the VM infrastructure, and see how it reacts. After this works properly, some of the nodes are disabled, and tests happen again to make sure the infrastructure works properly after a failure.

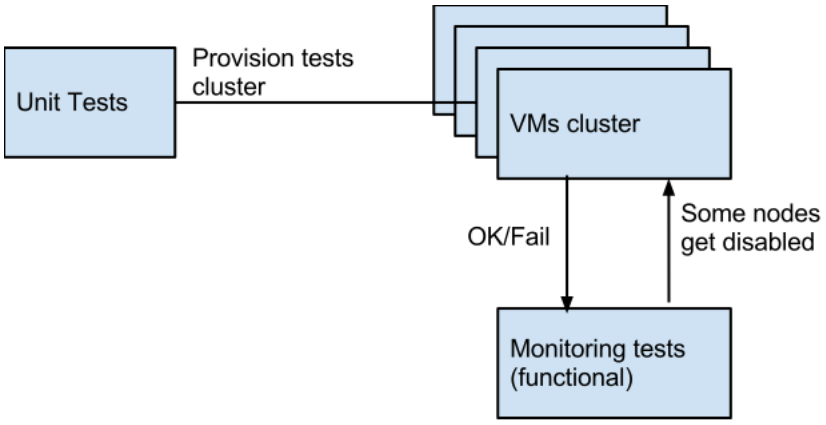

Figure 4.10: Monitoring testing cluster

4.7 Data storage

Currently one of the most commonly demanded features of a cloud is programmatic access to data. As opposed to the trend decades ago, storage silos only reachable through special protocols are no longer a valid approach in an interconnected world. Nowadays cloud data storage needs to be highly scalable, distributed, with no single point of failure and should support many concurrent users. Two very different data storage use cases can be considered.

4.7.1 Object Storage

Akin to S3, our deployment needs the ability to serve data objects of any size efficiently. These days data object backends need to support many concurrent reads and writes. Of course, these operations can modify the data, so our object storage backend needs to be *eventually consistent.*

Openstack recommends a S3-compatible replacement to AWS S3 that has an interesting architecture. All communication is done through a RESTful API to return objects. That allows users to run their queries using standard HTTP verbs such as GET, PUT, POST, DELETE. The URLs are structured in a hierarchical way as follows:

```
http://swift.example.com/v1/account/container/object
```

- Account is determined by the auth server
- A user should ask for containers in his own account.
- Containers are a convention to put objects or other containers inside
- Objects can be any kind of data, usually blobs

4.7.1.1 Proxy

A proxy will be the only interface with the outer world. Depending on the API requests, it will be intelligent enough to gather information from the appropriate backends and respond properly. This component will be stateless. If possible, two proxies should be deployed, with a DNS load balancer on top, for enhanced availability.

4.7.1.2 Ring

The ring makes sure partitions are replicated to physical locations on disk as per below, partitions subsection.

4.7.1.3 Partition

Any collection of data is a partition. A defined size is set for partitions, then three replicas of the partition are sent to disks in different zones to prevent data lost.
Every time a partition is modified, the *replicator* checks the hash of the other copies to make sure the rest were changed as well. If that is not the case, checks for timestamps are made, and the most recent changed partition is the one that gets replicated across other nodes. The following figure shows the described procedure.

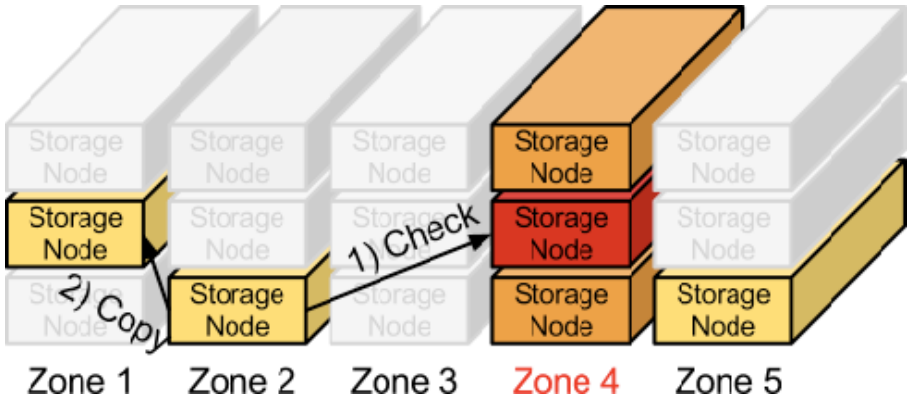

Figure 4.11: Data partitioning in Swift after modification

4.7.1.4 Accounts and Containers

Accounts and containers are SQLite databases that contain objects. These are the 'objects' at the level the ring operates. The ring performs its operations at this level and not shuffling objects across containers unless the request explicitly asks for that. In fact, moving accounts and containers around very often would degrade the service as mutex locks need to be acquired in order to move containers around, concurrent reads or writes in this case will easily timeout.

4.7.1.5 Object data store overview

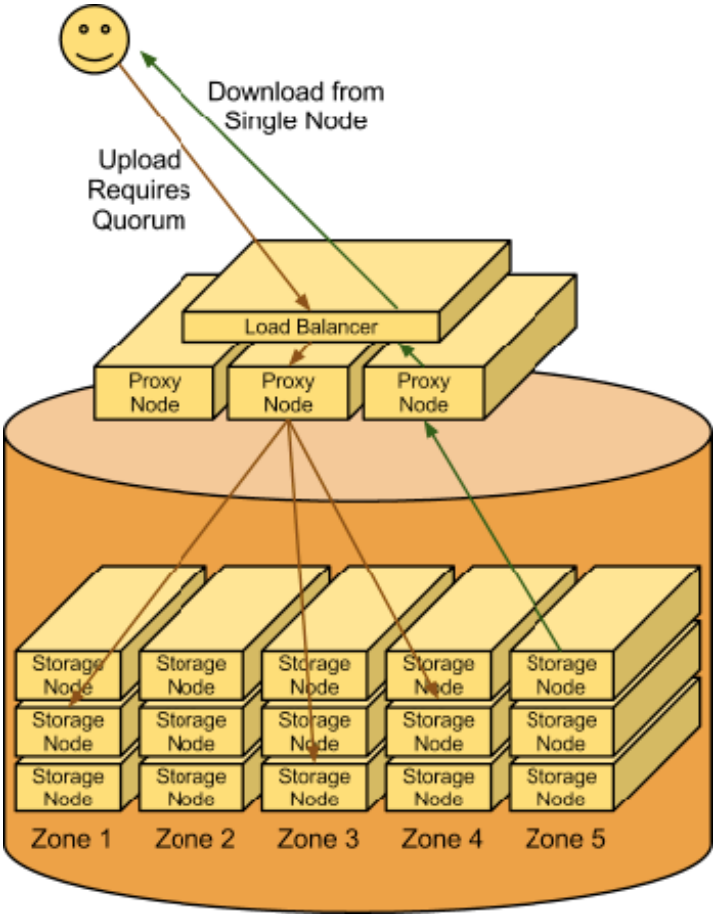

Figure 4.12: Overview of data storage system

4.7.2 Block Storage

Block storage is focused on providing a fast, consistent, concurrent volumes service to our virtualization infrastructure. One of the primary goals for this component is to be able to perform very *fast real-time I/O* as this will affect the experience of users. For instance, *Swift* objects are theoretically an option to use, but since the objects are meant to stay static, in practice the reallocation of these objects across the backends (partitioning and eventual consistency) will make our volume several orders of magnitude faster than with a storage system designed for this use case.

Snapshots of these volumes can also be an interesting feature to restore broken virtual machines. In some cases, including a component with this feature can be the difference between a system that is able to comply with national regulations and a system that cannot do so. There are many alternatives in this part of the architecture to the standard nova-volume, such as Ceph Block, GlusterFS, and others, but for non intensive I/O applications nova-volume will suffice. A special I/O intensive service could be provided with the help of these tools, but it would require a separate

deployment of the virtualization stack (Nova).

4.8 Parallel job execution

As the number of server grows up, sometimes the need to run a program, or perform any operation at all across several machines, just once, will appear, and doing it manually would be terribly inefficient for server farms with thousands of hosts. At that point, running a command over an SSH 'for-loop' does not scale very well either, but it can be the only solution.

Fortunately, there is a part of our setup, shared by every machine. Puppet, the configuration management system. All servers need to be configured and their Operating System images and Kickstart come ready with the configuration needed to connect to the masters explained in previous sections of this guide.

Marionette Collective (MCollective) can be our tool of choice for these kind of tasks that need to be run across many hosts. To provide a fast, parallel service, the way it works fits on a publish-subscribe paradigm. Here is an example of the output of an MCollective query made to 11 hosts.

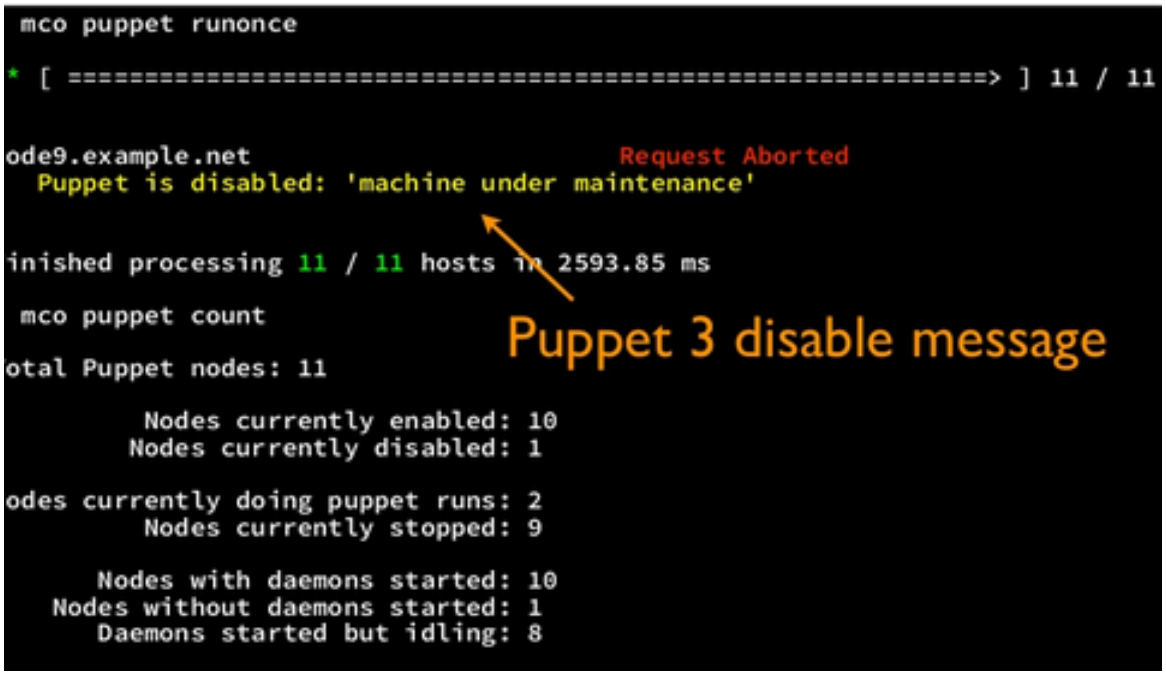

Figure 4.13: MCollective command line output

The way this is made possible is through three key components:

- A *client* that sends requests to any number of servers, using any plugins available, and receives data from these servers to be shown on a command line interface. A connector plugin that sets up a connection with the middleware needs to be configured before running any command. This is the tool system administrators will have to learn to use.

- A *middleware* that handles requests from client to server and from server to client. Normally this is implemented via an ActiveMQ/RabbitMQ message broker. Whichever option is used for the message queue in Openstack, can be reused here for simplicity.

- *Servers* who connect to the middleware through the same connector plugin the client uses. That allows them to receive messages from the broker in the middleware, perform the operation, then reply with a message containing whether they succeeded or failed.

By default clients send empty messages to discover a list of hostnames available. Nonetheless, there are custom plugins that allow users to fetch this information from other sources, such as a network topology, an artificial division of hosts depending on what is their use (Foreman, mentioned in this chapter has a discovery plugin for this).

A cluster of brokers should be created to allow MCollective to work efficiently and create subcollectives per network. This scenario would consist of several independent data centers but with a cluster of message brokers that allow MCollective to work properly across them. A common use case where this is needed, occurs when networks are in the same data center but separated for security reasons.

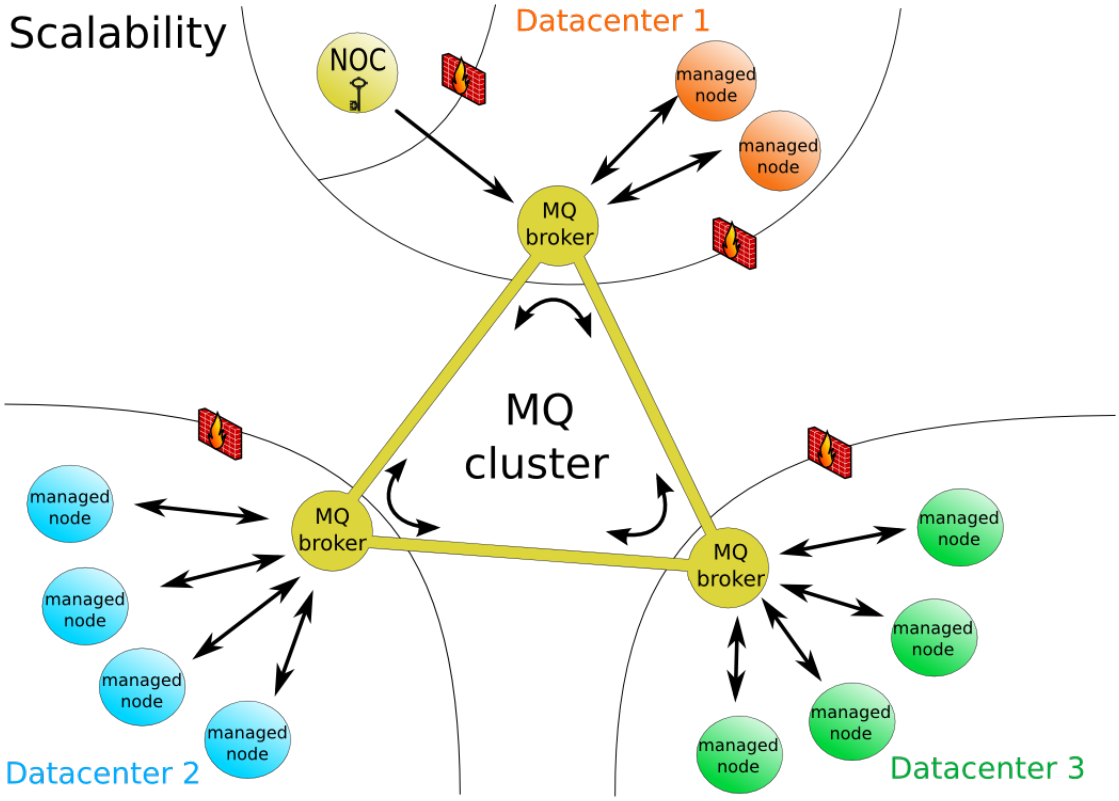

Figure 4.14: MCollective Message Queue Cluster

4.9 Contributions to the ecosystem

It will be needed at some point to perform power operations over the network on all of these systems. Usually, physical boxes have a secondary, always turned on, network interface, that is connected to the system BIOS and interact with the system at the lowest level. These *Intelligent Platform Management Interfaces* (IPMI) have a set of vendor-specific tools that can easily interact with, and a set of open source tools such as 'freeipmi' or 'ipmitool' that try to offer a common gateway and perform operations on any interface supporting this standard. IPMI network cards have a different IP and are always reachable regardless of the state of the physical box they are connected to.

Since Foreman contains an inventory of all nodes, physical and virtual, it is a very convenient way of indirectly interacting with these boxes. Personal contributions have gone a long way towards making remote IPMI interfaces visible in Foreman, as explained in chapter 'Tools', section 'Contributions to the Ecosystem'.

The main idea here is that since Foreman is the gateway for these operations, it should understand how to interact with any kind of machine. In the case of physical machines, it connects to a proxy that runs the actual 'freeipmi' or 'ipmitool' commands. Of course the success of this relies upon the vendors implementing the IPMI de facto 'standard' API, as no real standard exists for this yet.

Virtual machines are easier to deal with. A library called Fog provides a cloud API in Ruby so that the Foreman application simply needs to be aware of under which virtual appliance are the virtual machines (Openstack in this deployment, EC2, Google Compute Engine, etc... are also valid options), then run the appropriate call to Fog, and Fog will act as a translator between Ruby and calls to these APIs.

Summing up, Foreman is a passive agent that understands how to send power operation commands to any of our compute resources, be it virtual or physical.

4.9.1 Power operations in Foreman UI to appliance

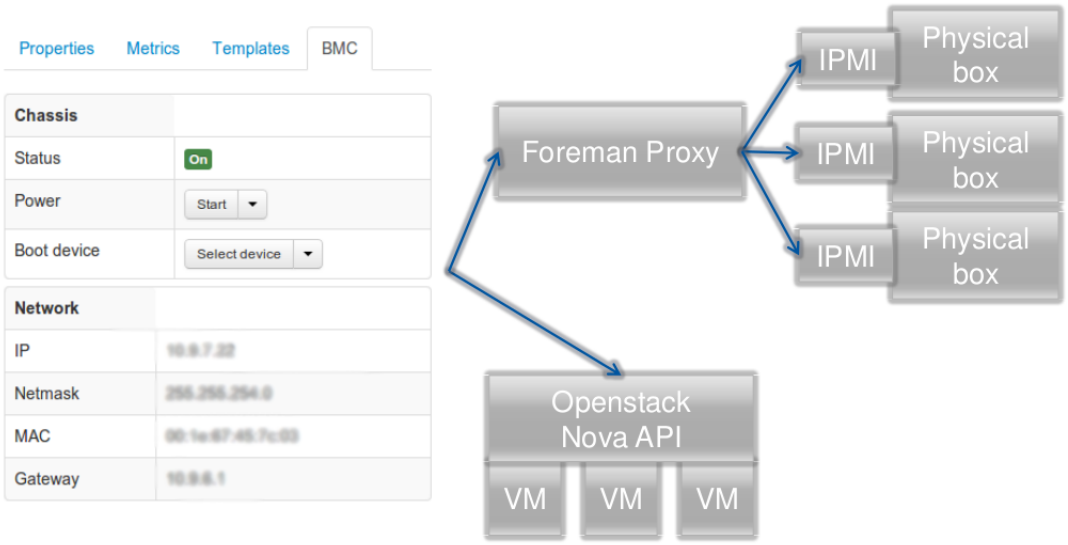

Figure 4.15: Foreman power operations common gateway

4.9.1.1 IPMI API through Foreman

Foreman is at the forefront of operations from the point of view of the people managing the infrastructure and for users.

Physical machines in most infrastructures have two network cards, one of them for standard Ethernet internet connection, and another one for IPMI connections. IPMI, *Intelligent Platform Management Interface*, is a vaguely standardized interface protocol started in 1998 as a joint venture of Intel, Cisco, Dell, HP, and NEC. It is implemented at hardware level and runs independently from the operating systems. This is intentional, because one of the main features of IPMI is that it lets the user run power operations, and interact with the BIOS, remotely. IPMI works through a microcontroller in the motherboard, called the BMC, baseboard management controller, which is connected to a NIC, a serial port, and a series of buses that connect it, the power supply, the BIOS and so forth.

Even though there is a standard defined by leaders in the servers industry, not all vendors implement it the same and therefore it is hard to make tools that work in all vendors. This is why the list of IPMI actions that are available through Foreman is restricted to only power operations and choosing which device to boot from, because these are universal actions known to easily work in all implementations.

My contribution exposes IPMI operations through the UI, as per figure 4.15. Foreman issues HTTP calls to the Foreman Proxy, which in turn runs these IPMI commands using an UNIX tool called ipmitool. This allows users to seamlessly perform power operations (on, off, soft reset, acpi reset, cycle) and selection of boot device (PXE, disk, safe mode, cdrom, BIOS).

Since users cannot perform operations on thousands on hosts through the UI, I

made another contribution that opens an API for users, so that using their credentials to Foreman, they can perform the same UI operations by issuing HTTP requests to Foreman's API. This has allowed people to write a command-line interface program that connects to Foreman, and run a command across many nodes. For instance, if the user wants to reset all Hadoop nodes that are in testing mode, the user can simply run:

```
foreman-power-control --hostgroup=hadoop --environment=test reset
```

This is a powerful feature for administrators who otherwise would have to write a wrapper for ipmitool or other UNIX tools, deal with thousands of passwords (one per IPMI interface). Instead, now administrators have this option and if they use Foreman they can perform power control operations across defined groups of nodes in a much better way than before.

This contribution was merged upstream and attached to these links you can find the code.

- Source:
  - BMC/IPMI commands - https://github.com/theforeman/foreman/pull/556
  - API - https://github.com/theforeman/foreman/pull/762

4.9.2 PuppetDB Foreman Plugin

As per 4.4 Configuration Management, Puppet will be the tool of choice for this cloud. When resources are configured, packages are installed, or any other operation is ran through configuration management, a report is generated. These reports are stored in a database called PuppetDB. This database stores the content of all reports, so it stores the *current state of all nodes in the infrastructure*. As opposed to Foreman, which stores the desired state of all nodes, PuppetDB stores the actual state. This allow users to write programs using real time data of each node. The data that is stored in PuppetDB include facts about the machine, such as the version of the GNU/Linux Kernel it may be running, the latest user that logged in, and others.

When any machine is destroyed, PuppetDB obviously will not receive reports from that machine. Nonetheless, from the PuppetDB side, it is impossible to know whether the machine has been removed from the infrastructure, or if it is just taking longer than usual to submit a report from that machine. This wastes disk space per each node who has been removed from the infrastructure but not from PuppetDB.

I contributed back to the community with a plugin that allows Foreman to connect to PuppetDB and let it know when a machine is actually removed from the infrastructure. Since Foreman stores how the infrastructure should look like, whenever a host is removed in Foreman, a series of queries are sent to PuppetDB, which trusts Foreman as an authenticated source, and these queries let PuppetDB deactivate a node and remove its information from the database, thus not wasting disk space. An option to deactivate nodes if they have not supplied information to PuppetDB in a number of days is also provided. This has become a common tool used in several deployments on similar cloud infrastructures.

- Source - https://github.com/cernops/puppetdb_foreman

4.9.3 Foreman MCollective Discovery

Marionette Collective, as explained in 4.8, is the last tool mentioned on this chapter used to run operations across many hosts. Via broadcast signals to the middle-ware, the default discovery system realizes the number of hosts in the network that are available to receive these operations. Of course, sometimes the network, or other problems, might not allow MCollective to discover all nodes the user wants to run their operations against. Using databases, such as PuppetDB, network topology databases, or other solutions were available when I started this project.

However, these databases might not have classifications by groups of hosts that are used for a certain thing, for instance, nodes used for storage. Foreman hosts are usually sorted depending on which is their role in the infrastructure, say, hadoop processing nodes.

This contribution allows users to search in Foreman for the name of the hosts it should run upon. An example of syntax to search in Foreman instead of the default search:

```
$ mco xxxxx --dm=foreman --do='params.owner = owner1 AND ip=127.0.0.1'
```

That would perform a query against Foreman that returns a list of hosts. MCollective sends messages to the publish-subscribe middleware using that list of hosts, and runs whichever program the administrator wants to run, and returns the output to the client.

- Source - https://github.com/cernops/mcollective-foreman

4.9.4 Puppet-lint code outside scope

In order to enhance the quality of Puppet code in this deployment, a hook that is run upon every code commit checks whether the style of the manifests (configuration) code is how it should be according to the official style guide. Besides, this contribution helps identifying parts of code that trigger a bug and should not be run by the Puppet masters.

The bug triggered is Puppetlabs #18282 (http://projects.puppetlabs.com/issues/18282). Essentially, the Puppet masters read code on the top level scope of a manifest. Then, when another node requests a catalog compilation for its manifest, even if there is no top level scope code, the Puppet master is unable to realize that, and uses the top level scope code from previous catalog compilations. This problem can be solved by forcing the Puppet master to reload the process every time a compilation is requested, but that could be very costly. Instead, it is better that the code that is in the manifests does not contain anything on the top level scope, as this is accepted as best practice in the community. Everything that is not within 'class' or 'define' brackets, is top level scope code.

```
class foo::bar { file{'/etc/passwd':} }
include ('mymodule')
```

In this example, 'include ('mymodule')' is in the top level scope. That would install the module 'mymodule' on every machine until the Puppet master is restarted, since top level scope is shared across nodes due to the aforementioned bug.

This contribution builds upon puppet-lint's lexer. This lexer reads a Puppet manifest (configuration declaration), tokenizes it, and attaches extra value to each of the tokens. For example, tokens within square brackets, are easily accessible as they an inner level of abstraction.

Once the Puppet manifest is parsed by the lexer, all tokens that are in 'class {}' or 'node {}' definitions, are indexed and removed from the full array of tokens. At this point only the tokens on the top level scope remain. Each of the tokens on the top level scope is checked to see if they are anything that is not a comment, whitespace, or carriage return. In that case, that is bad style and triggers the aforementioned bug, so puppet-lint throws a warning. Using this warning as a check-hook on a git repository kept people out from committing code that could cause big damage.

- Source - https://github.com/rodjek/puppet-lint/pull/223

4.9.5 Foreman user groups linked to LDAP groups

Lightweight Directory Access Protocol (LDAP) is a protocol widely used by large enterprises to keep track of hierarchy, levels, roles, and groups within an organization. This allows authentication systems to have a source of truth to rely upon and have all services in the organization work with the same credentials.

Moreover, in a private cloud setting, there are groups that will have more privileges than others in terms of quota, machines they should be able to access, accessible resources, and others.

Before this contribution, Foreman had a concept 'User group' that required manual intervention from the administrator to replicate the hierarchy of roles, access, that LDAP provides.

Contributions to Foreman helped bridging LDAP user groups with Foreman user groups in an automatic way, importing users automatically, and setting the roles for these users automatically. In fact, users do not even need to sign up in Foreman if they belong to an LDAP user group that already has representation in foreman. They log in with their credentials, usually shared for all services in the organization, and they already have their preferences set.

- Source - https://github.com/theforeman/foreman/pull/529

4.9.6 Openstack Nova Power Operations Support

As per 4.9.1.1, Foreman is supposed to be an entry point for all power operations, regardless of the backend, virtual or physical. Foreman is smart enough to know how to handle power operations issued to different backends, which should be transparent to the user. A few paragraphs above the physical boxes case was covered using IPMI.

In this contribution Foreman is able to run power operations against an Openstack Nova virtualized box, on any tenant.

Nova offers an API to interact with the virtual machines directly. Nonetheless, users should not need to issue commands to this API. Instead, Foreman makes use of Fog, a Ruby library that wraps cloud libraries such as vSphere, Google Compute Engine, and of course Openstack.

Given this wrapper, Foreman needs to check on incoming power operation requests which kind of machines are Openstack Nova virtual machines. Whenever that is the case, Foreman uses Fog to issue whatever call the user asked for to the Openstack Nova node. The node returns a code telling whether the operation was successful or not.

This behavior is exposed through the UI and the API. A screenshot of the UI for these Openstack nodes follows:

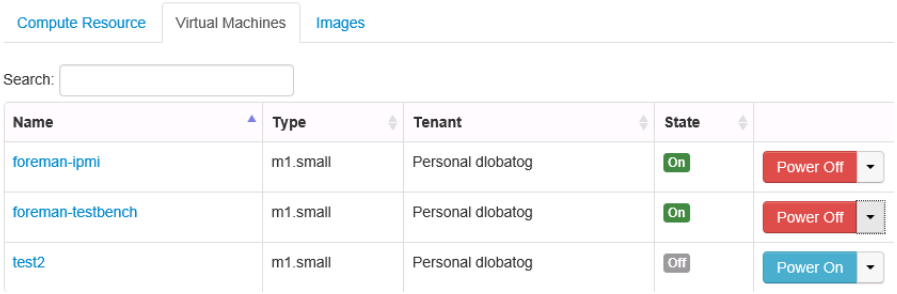

Figure 4.16: Openstack Nova UI for Power operations in Foreman

- Source - https://github.com/theforeman/foreman/pull/845

4.9.7 Authentication/authorization Refactoring

Foreman should be accessible through several authentication solutions, such as Single Sign On services, OAuth, and others.

Before this contribution, Foreman was able to authenticate using different models for different kinds of access. OAuth and Signo were available for API access but not for UI access. Delegated load balancer authentication using REMOTE_USER was available for the UI but not for the API. This authentication consists on reusing a cookie provided by a Single Sign On service, sending this cookie to the load balancer, and have the load balancer authenticate the user to the application. The load balancer sends a request to Foreman containing the header REMOTE_USER and Foreman trusts requests containing this header from a set of IPs. This allows users to avoid authenticating in each request, and simply reuse a cookie provided by the SSO service until it expires.

That was the main motivation behind this contribution, but since authentication methods were different for the API and for the UI, there was many duplicated code, and the user could benefit from knowing it can use any authentication method to request anything to Foreman. This resulted in a big refactor of all authentication methods into a single point of entry. Following Object Oriented Programming patterns the result is modular and it is easy to detect the parts of the code that

are relevant for every authentication method. A single point of entry detects what method is being used, and then authenticates in whichever way is more fit depending on the kind of request, API or not.

- Source - https://github.com/theforeman/foreman/pull/822

4.9.8 Foreman Parameters API

Parameters in Foreman are used to specify particular things about a host, group of hosts, and others. Parameters are usually retrieved at the host level to see if a node should be treated in a special way. When these parameters are applied to a group of hosts all hosts in that group inherit the parameter.

These parameters are simply a hash, with values such as "firewall -> on, kernel-params -> none", etc.. Values in these parameters are mere key-value pairs that do not have any effect in Foreman. However, they can be useful at the configuration step, or at any other step in fact.

At the configuration step, when the node asks for a configuration to the Puppet masters, the Puppet masters check the value of these parameters in Foreman and can provide a customized configuration to a node based on that, for example, turning off the firewall on a particular node.

Sometimes, other applications that are not the Puppet master might want to access these parameters for whatever reason. A user could set a host parameter "purge -> yes", on all nodes that the user wants to purge automatically. Then the program the user has written to purge the hosts, needs to check these parameters on all nodes owned by the user.

This contribution exposes these parameters through a REST API so that users can build their applications on top of their parameters, and if they want to remove parameters from a host, they could programmatically do so.

An additional helper to reset all parameters and leave a host without any parameters was added in this contribution for convenience for the users. In fact, removing parameters is an expensive operation consisting on at least one call to check all the parameters, and another call for each of the parameters that actually deletes each of them.

- Source - https://github.com/theforeman/foreman/pull/461

4.9.9 Minor Contributions

See other contributions at: https://github.com/theforeman/foreman/pulls/eLobato

Chapter 5

Deployment

5.1 Highly Available MySQL

A large part of our cloud deployment relies on a highly available SQL database. Chapter 4 contains some of the reasons why MySQL is a sensible choice for an Openstack deployment.

It is possible to engineer a MySQL deployment focusing on high availability and overseeing the particular needs of Openstack. If this proves to be not performant enough in a deployment, measures such as the number of writes/reads per second, table vs index access, will allow to customize a highly available deployment to whatever performance needs are required in a particular scenario. Several options will be explained in this section, leaving the choice to the team deploying the cloud.

5.1.1 MySQL cluster

This is an ACID compliant, distributed multi-master architecture. Like the rest of the options explained in this section, its most important features are high availability and replication.

5.1.1.1 Types of nodes

NDB management nodes are responsible for setting up the cluster. They control what responsibilities does each node have, can perform power operations across a cluster, and can actually access the data since they are connected to the data nodes if needed. However, normally SQL nodes are used for this purpose.

Data nodes contain the data and are automatically sharded. Cross-shard queries work properly with the use of another kind of node (SQL nodes) that acts as an arbitrator.

SQL nodes connect to the actual data storage. Since these nodes are an extra layer on top of the data nodes, they are not strictly necessary, but they can provide a familiar interface for data storage and retrieval. The NDB can connect to the data nodes itself through its API.

Data, but not metadata, are replicated between members of a node group guaranteeing that at least two data nodes keep the same copy of a piece of data. Replicating clusters can be done manually and is an excellent measure to maintain low latency across different network zones, or simply to prevent against catastrophe

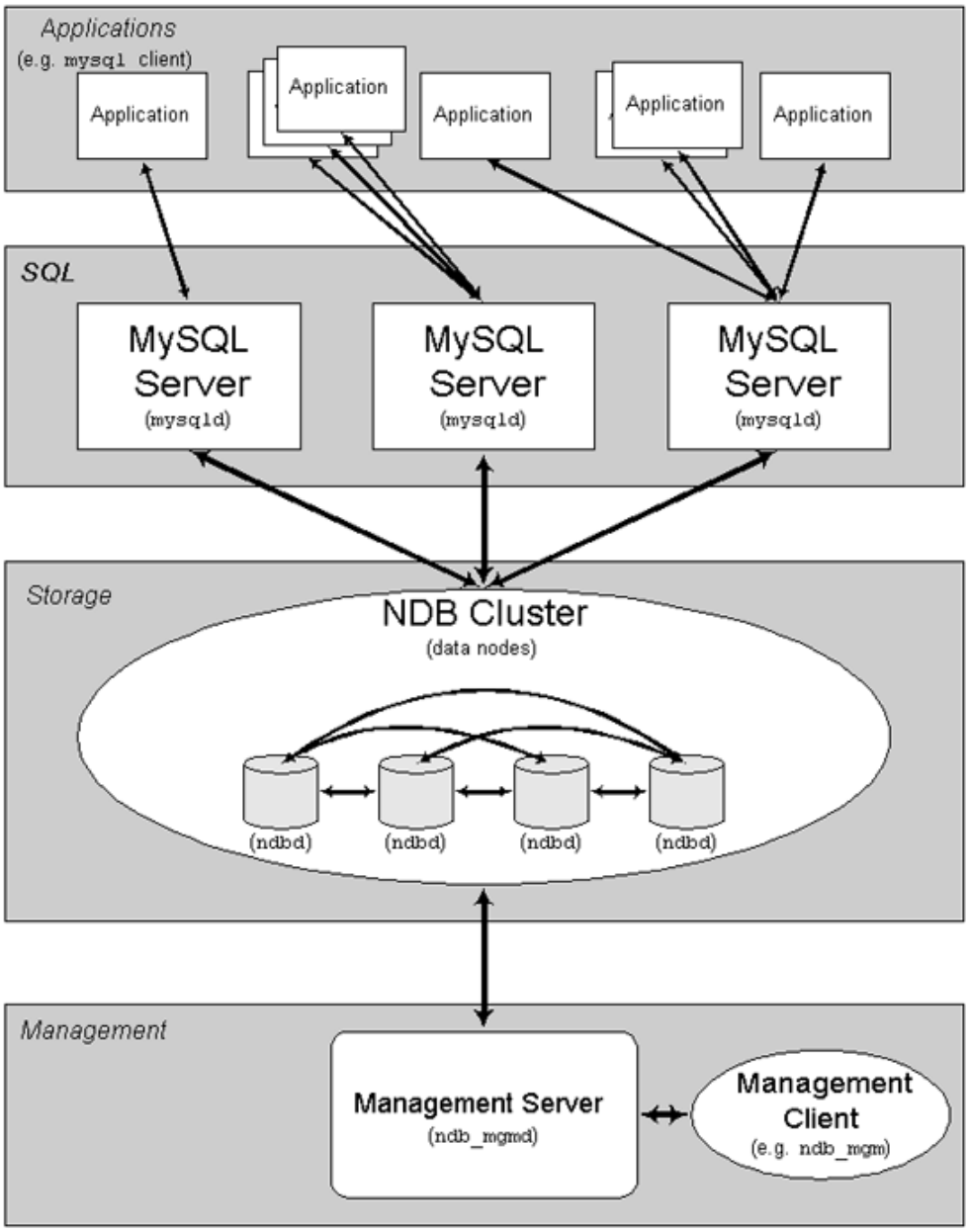

Figure 5.1: MySQL cluster with NDB nodes

5.1.2 MultiMaster replication manager

A highly available solution that employs commodity hardware and requires very little setup. Only two data nodes, two IPs, and a monitoring server are needed.

Two (or more) data nodes will store the data and respond to requests made to the IPs.

The monitoring server checks the availability of the nodes. Whenever any of them go down or underperform in terms of writes or reads, IPs get moved to a different node. In order to get better availability, two monitoring nodes should be used.

In case of high demand, a new IP could be brought from a pool of floating IPs for reads, writes cannot be load balanced. First the monitoring node takes note of that, then it is deployed. These IPs can be load balanced with any common software such as HAProxy or Apache mod_proxy_balancer. In fact, this is a feature that will very easily make for an easier -clients will only have to point to a single, non changing IP- and a more performant deployment because of the balancing.

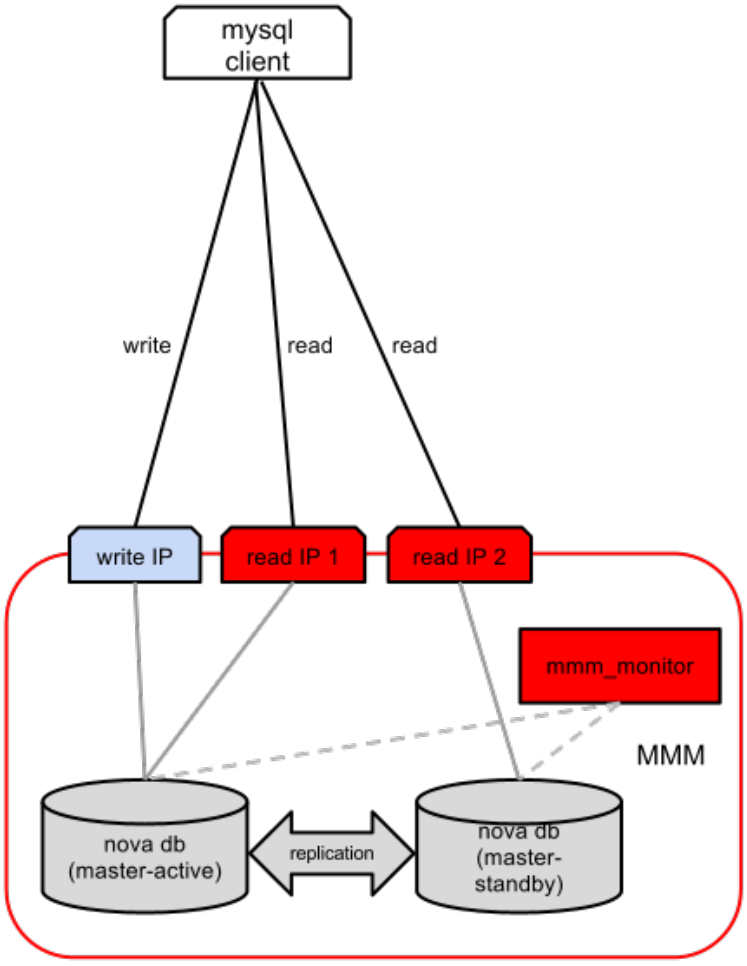

Figure 5.2: MultiMaster replication manager concurrency

5.1.3 Galera cluster

Galera is a plug in for InnoDB (MySQL's storage engine). As such, it tries to overcome some of the problems the MySQL standard cluster has.

MySQL offers replication by default, but some issues such as dealing with multiple writes to the same piece of data in different masters. What we gain by using Galera as a plug-in for InnoDB is that all clients can write to or read from any server, with guaranteed consistency.

Depending on the version of NDB (MySQL Cluster), adding online -no downtime- nodes to the cluster can be impossible while Galera offers this from the very beginning. NDB does not perform as well as Galera on commodity hardware and less so in specialized hardware. See benchmarks (http://codership.com/content/whats-difference-kenneth)

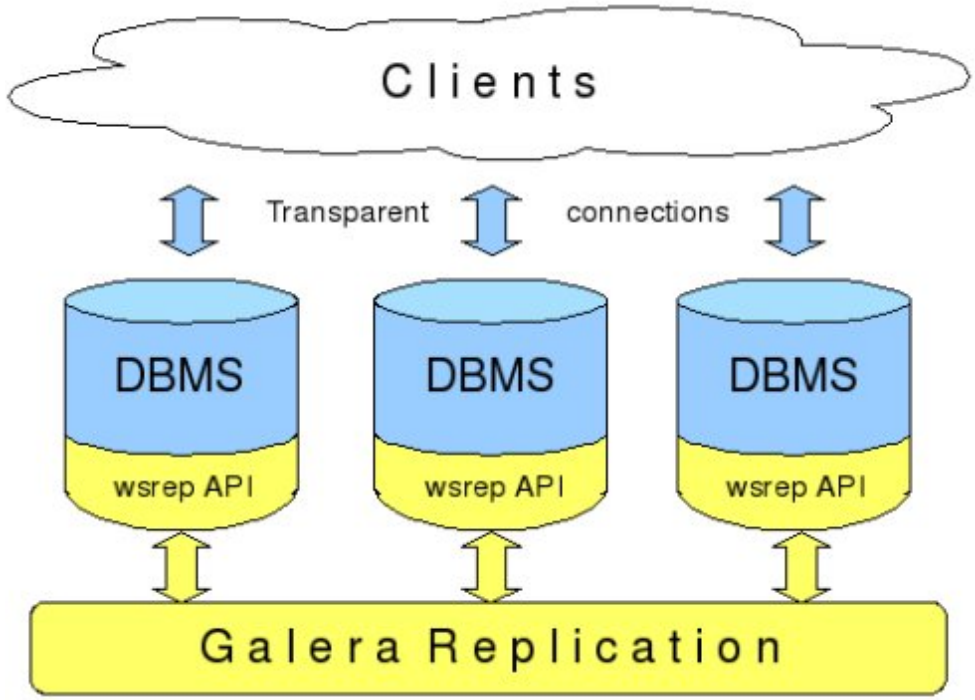

Figure 5.3: Galera replication

5.2 Highly Available RabbitMQ

Message queues are mostly focused on speed to process large amounts of data. For that very reason, messages cannot be stored on disk and are usually served from memory. This poses a whole different set of challenges compared to databases whose data is not subject to these restrictions.

A regular *cluster* of RabbitMQ queues, load balanced, is a solution good enough for deployments where losing some messages if one of the server breaks is not a problem. If one node in the cluster is broken, other nodes can still serve requests,

but the contents that node had in memory are lost. In this service, this could lead to problems like having allocated disk and processing power to a virtual machine, but not a network address and the virtual machine would be unreachable.

Active/passive setups, where passive nodes (slaves) take on the work the active node (master) had, can be a good option, but the queue might be down for some time until the passive queue starts up.

Active/active deployments are fundamentally similar to active/passive deployments, but the slaves are mirrored queues of the master one. The policy for picking a slave node when needed is to pick the eldest one. This allows no downtime, and a slave which is as close as possible to synchronization with the master.

After the slave -from now on the master- has taken on the old master, it assumes the clients have disconnected and reissues all pending messages to consumers.

Durable queues can also be considered but are not needed in a setup where old requests are usually unimportant. A possible way to improve availability in this setup is to monitor with Nagios incoming queue requests and see if they are not stored on the durable storage on failure.

5.3 Configuration Management Masters

As per the decision taken in chapter 4, Puppet will be the tool of choice. A good way of narrowing down this problem is to think of the deployment as a *master-client* setup of nodes that are going to serve and receive *HTTPS requests.* Given these constraints, we can focus on how to scale these kind of systems in general and then explain what is the role of Puppet in a scalable deployment.

5.3.1 Certificate Authority

In a master-client setup, the master will need to identify the clients in some way to provision them with the right configuration. Nodes requesting configuration using rogue host names will be a security threat to this environment.

For this reason, a certificate authority, that will be shared across masters needs to be setup. The course of action will be the following:

1. Client comes online
2. Client runs Puppet agent, which generates a Certificate Signing Request (CSR) and sends it to the master
3. Master sends this CSR to the Certificate Authority (CA)
4. CA signs it
5. Master trusts the node is who it claims to be

Setting up the CA server is outside of the scope of this document, but judging by the requirements, several CA servers will need to be setup in such a way where the

passive servers are actively mirroring the master CA server so that if it goes down one of the passive nodes can quickly replace the broken master CA.

5.3.2 Multi site scalability

Much of the lag that inevitably comes from having agents far away from the clients can be lowered by spreading the masters across many physical data centers. This, however, will not make the first run, in which clients have to create the aforementioned CSR and get it signed faster.

A DNS server can use BGP Anycast routing to announce a different IP address depending on which part of the world the client is. This IP will have to be the IP of a load balancer -there can be several- that is behind a pool of Puppet masters.

In the end, the request will get to the Puppet master and assuming the certificate is not checked at the Puppet master every time, by caching it, it will assure the deployment a low response time and enough availability. Worst case scenario the node will have to be checked against the CA, but that should be a rare case, and as such the CA should not be overloaded.

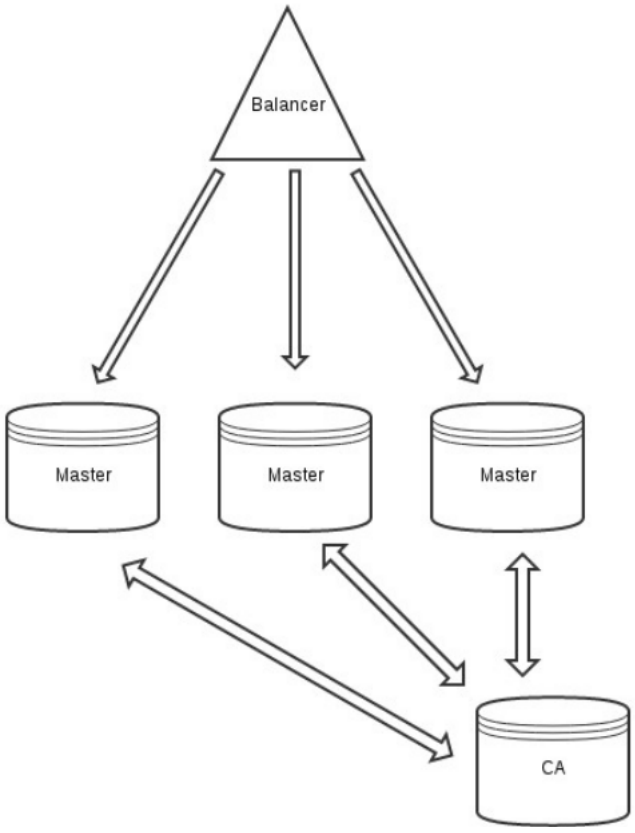

Figure 5.4: Puppet multi master setup

Above is a picture of the last layer that the request goes through after it being assigned a load balancer.

5.3.3 External Node Classifier

As explained in the previous chapter, the ENC will let our users place their hosts in a host group which comes with a predetermined configuration for that host. This makes the job of creating clusters of machines very easy. This information is coming from the Foreman in a YAML parsable file.

5.3.4 Splitting up services

Applications such as Foreman or even Puppet offer more than one single thing. Because of that, it is worth it to separate the infrastructure into different services. Part of the reason why is that upgrades to the different services can be simply tested on the nodes relevant to that service, instead of testing things across the whole infrastructure.

For instance, in Foreman, there are three essential components

- External Node Classifier
- API
- UI
- Reports processor

It is likely that if any of these parts is broken at some point, the others are unaffected. For this very reason, we can split the service up behind a load balancer, assigning a different port to the different services, and keeping a copy of Foreman running behind these ports in a pool of servers. It will make the configuration slightly more complicated as opposed to simply keeping standard Foreman running under the balancer, but the design pays off in terms of scalability and availability.

```
---
classes:
    common:
    puppet:
    ntp:
        ntpserver: 0.pool.ntp.org
    aptsetup:
        additional_apt_repos:
            - deb localrepo.example.com/ubuntu lucid production
            - deb localrepo.example.com/ubuntu lucid vendor
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.example.com
    mail_server: mail.example.com
    iburst: true
environment: production
```

Figure 5.5: Sample External Node Classification output

Similarly, the Puppet masters can be broken into two different subsets, batch masters, which respond to background (batch) requests, and interactive masters,

which respond to user triggered Puppet runs. Background runs will be the majority of them and the system can be designed to better handle the load in that case, and possibly adding more features on the nodes that will return some output to users.

5.4 Modules workflow

Setting up git repositories is a seamless process, and many backends for this are available, so this is a topic that does not need to be covered here.

Nonetheless, a scalable process to provision the aforementioned Puppet masters with modules to configure their clients is needed.

5.4.1 Naive solution

A naive solution can be as simple as having a single git repository containing all modules. Each of the branches can be thought as a different environment, such as production, testing, development, etcetera. The masters will pull from this repository each of the branches to construct the environments locally. This would work relatively well if there is only one master pulling this information.

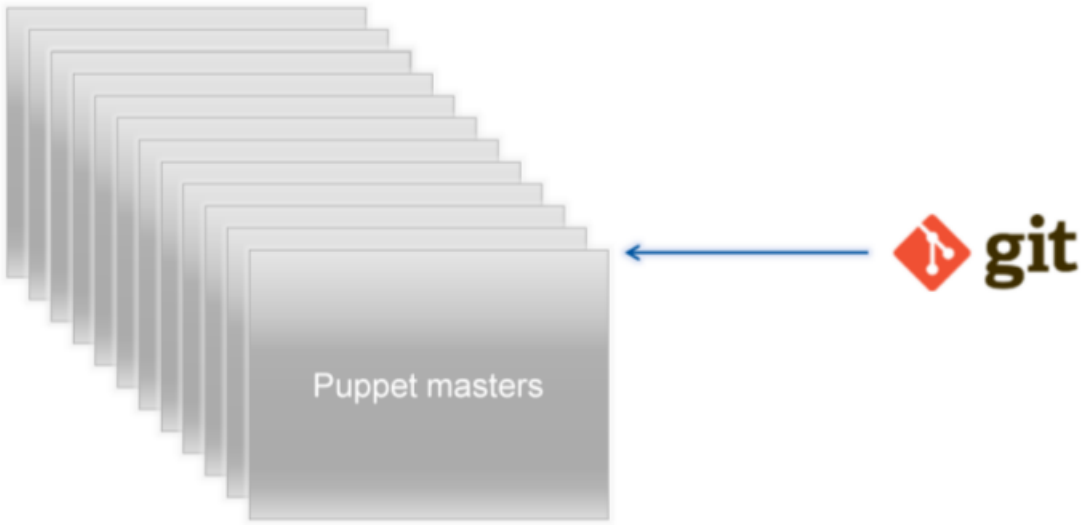

Figure 5.6: Simple git pull from puppetmasters

Unfortunately, this will not scale in a system that requires more than one master to configure the infrastructure. One of the reasons is that it is hard to synchronize the *pull* operation in all the masters. That means a client request could be hitting an *unsynced* master. If this client had recent changes that have not been propagated to all masters, this would revert the configuration of this particular node to a previous state, which is not acceptable.

One of the major issues to deal with is that nearly all operations in git cannot be concurrent for consistency reasons. For instance, if client A and client B want to push a valid change to the repository at the same time, the server needs to process the requests one at a time to ensure the changes are truly valid. This poses an important problem when it comes to synchronizing all masters to a single git repository, because the global state may vary between pulls from different masters.

Another issue to deal with in such an environment is that maintaining a production environment can be complicated. Either a lot of trust needs to be put in people who are writing the manifests, which is not feasible, or a verification partner - which can simply be the team deploying this setup - reviews the manifests and moves them to a production environment. The former is unrealistic, and the latter is inefficient at best.

5.4.2 Final solution

It comes clear after analyzing the process that a central source of truth needs to be concurrently provisioning all masters with a set of manifests per environment. Since the git server is unable to do so, a proxy can do it. This proxy needs to:

- Create several environments based on git branches
- Send this information concurrently to all masters

For the first task, one could argue that a single repository for all configuration needs might not be the best choice given some teams want to hide configuration secrets from other teams. In any case, the proxy should be able to create environments from either one or several git repositories. The main advantage of doing so is to avoid the inconsistencies mentioned in the previous section.

The second task can be easily done with a clustered file system such as GFS2 from Red Hat, but such systems cannot handle different availability zones for multiple masters easily. Setting up everything on the same availability zone is a bad practice because in case of hardware or power failures, clients will not be able to get the latest configuration.

Since high availability needs to be a key part of the design, the proxy can simply send the information concurrently to the masters which can be located in different networks and availability zones. pssh (Parallel SSH) can certainly do the job, but it is rather inefficient to keep on pushing the whole modules repository/ies, which can be large in size. Instead, rsync, utility software both available in Unix and Windows systems, will be more efficient since only changes will be sent across the network instead of actual files and directories.

This is a way of mimicking git's pull operation, which sends changes instead of files across the network, without having the git server handling such concurrency.

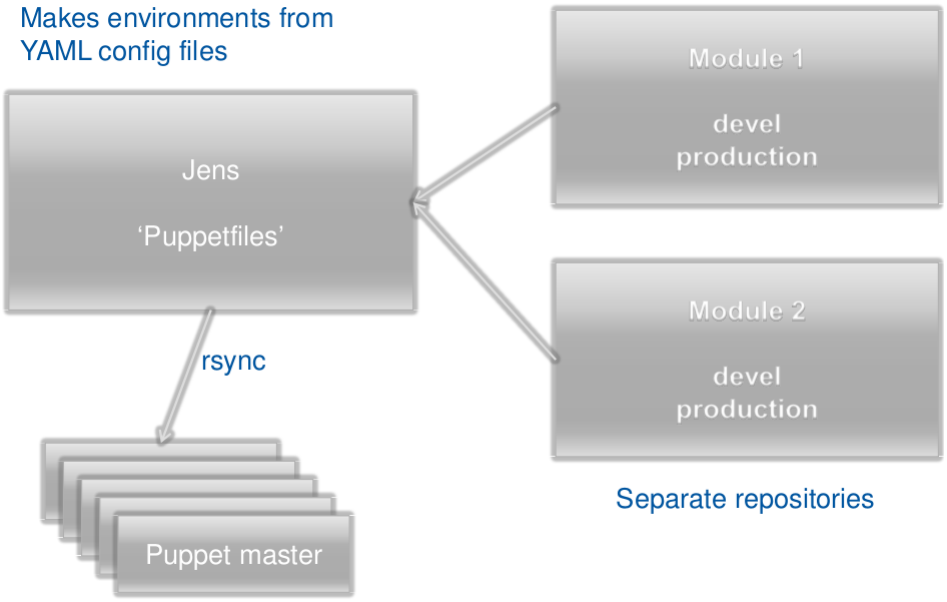

Figure 5.7: Advanced Git synchronization with masters using rsync

5.5 Auto scaling

Automatic scaling is a concept that involves systems scaling up and down their resources depending on their *load*. This service is usually tightly coupled with the virtualization product used on your cloud, although open APIs make this simpler than in the past.

In practice, auto scaling decisions are usually user defined policies, although there are some research lines investigating auto scaling policies that are computer generated. These policies usually do *checks* on a pool of servers and act accordingly.

Checks that trigger auto scaling can be varied, from simple health checks, to predefined conditions such as expecting more users during the weekend and hence increasing your cloud resources, or checking how steep is the CPU utilization graph to remove cloud resources when the slope is negative.

At the moment, Amazon's solution for auto scaling, which normally sets the example for other cloud services involves gathering metrics with the CloudWatch service, and use these metrics to trigger orchestration of machines on the CloudFormation service. Our cloud will employ Openstack Heat, which is API-compatible with CloudFormation, and Openstack Ceilometer as the gatherer of metrics that are used to trigger scale changes.

Following these lines there is an example of how to use Heat templates to orchestrate the creation of a node, providing examples from the configuration of a MySQL node.

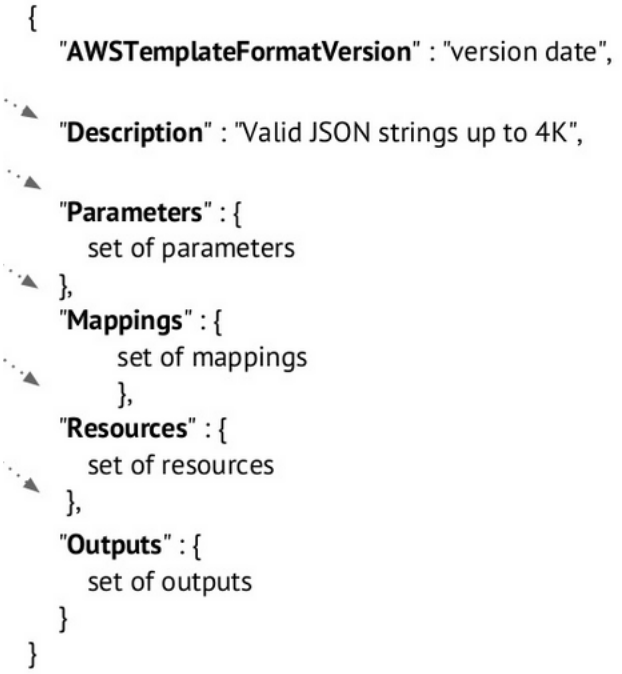

Figure 5.8: Heat template description

- Parameters: Optional parameters to be considered when the template is executed.

```
- "DBName": {
      "Description" : "The database name",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "64",
      "AllowedPattern" : "[a-zA-Z][a-zA-Z0-9]*"
  }
```

- Mappings: Allow to choose a specific attribute depending on the parameter value. In the example, depending on the value of parameter RegionMap, the machine would be created in whichever availability zone, using an OVF image defined by the user.

```
- "Mappings" : {
      "RegionMap" : {
          "us-east-1" : {
              "OVF" : "ovf-76f0061f"
          },
          "us-west-1" : {
              "OVF" : "ovf-655a0a20"
          },
          "eu-west-1" : {
              "OVF" : "ovf-7fd4e10b"
```

```
        },
        "ap-southeast-1" : {
            "OVF" : "ovf-72621c20"
        }
    }
}
```

- Resources: Contains an array of defined computational resources. For instance, two machines can be defined within Resources, including the kind of machine, and the Userdata (Kickstart) that will run on them.

```
- "MySqlDatabaseServer": {
    "Type": "Openstack::Nova::Instance",
    "Size": "m1.small",
    "Hostgroup": "mysqlnodes/production",
    "Owner": "Daniel Lobato",
    "Userdata": {
        .....
    }
}
```

- Outputs: Define values that users might want to get back when they run '*cfn-describe* stack'. The example shows a public IP of one node to return.

```
- "PublicIp": {
    "Value": {
        "Fn::GetAtt" : [ "MySqlDatabaseServer", "PublicIp" ]
    },
    "Description": "Database server IP"
}
```

5.5.1 Diagram Heat template <-> Cloud (Heat engine)

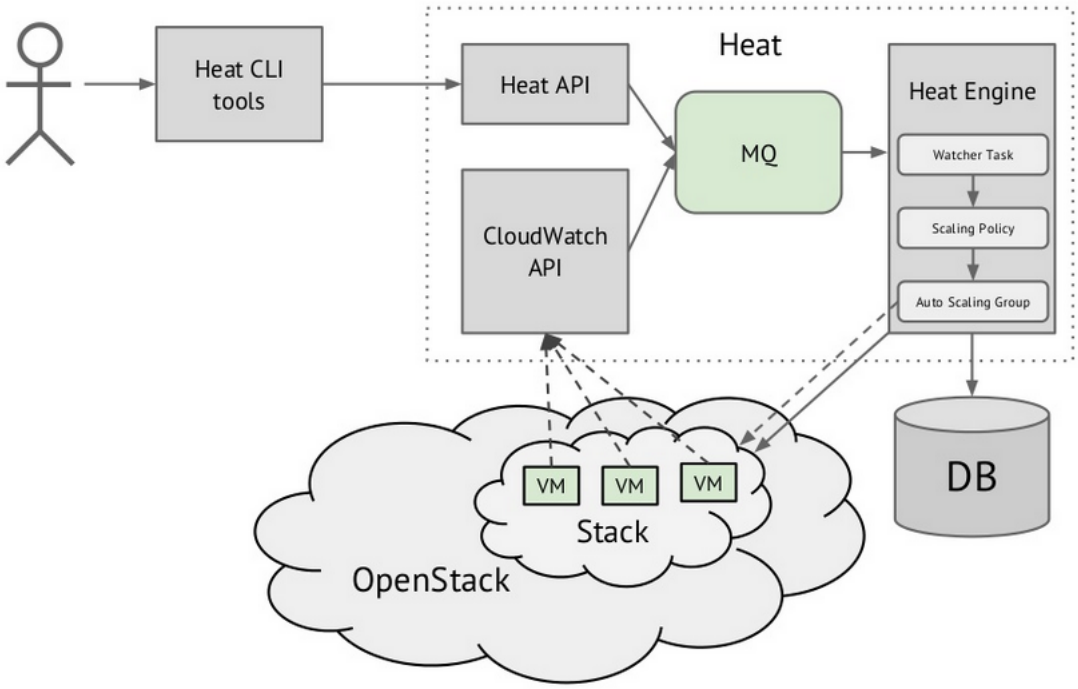

Figure 5.9: Heat Architecture

5.6 Openstack Infrastructure Topologies

The topology of Openstack is highly based on sharing nothing by default, and sending many messages across the architecture. Please keep in mind the virtualization figure in chapter Tools, section Virtualization for the explanation. Basically, the key insight to keep in mind is that nodes running *nova-compute* and nodes not running nova-compute (*nova-scheduler, nova-api*, etc...), should be physically separated.

Conceptually, three types of Openstack nodes come to mind:

- Endpoint node: These nodes usually run *nova-api*, possibly *Neutron* (load balancing as a service solution) and any kind of service that interfaces with the exterior. These nodes should be redundant, redundantly load balanced, to provide a decent uptime for users. They will use the public network very heavily compared to the other nodes.
- Controller node: Services such as the Openstack Dashboard, queues, monitoring, fall under this kind.
- Compute node: Hypervisors (hosts for virtual machines) will be created in these kind of nodes. If *Neutron* is not set in place, these nodes will host nova-network and provide network capabilities to their VMs, so all three networks (Private, Public and Management) will be used equally by these nodes.

In our deployment, these nodes will be as separated as possible as it can be seen on the next figure. Of course, this only needs to be the case for a production environment. Test environments can mix the three services for a quicker deployment, where no requirements of consistency or availability are needed.

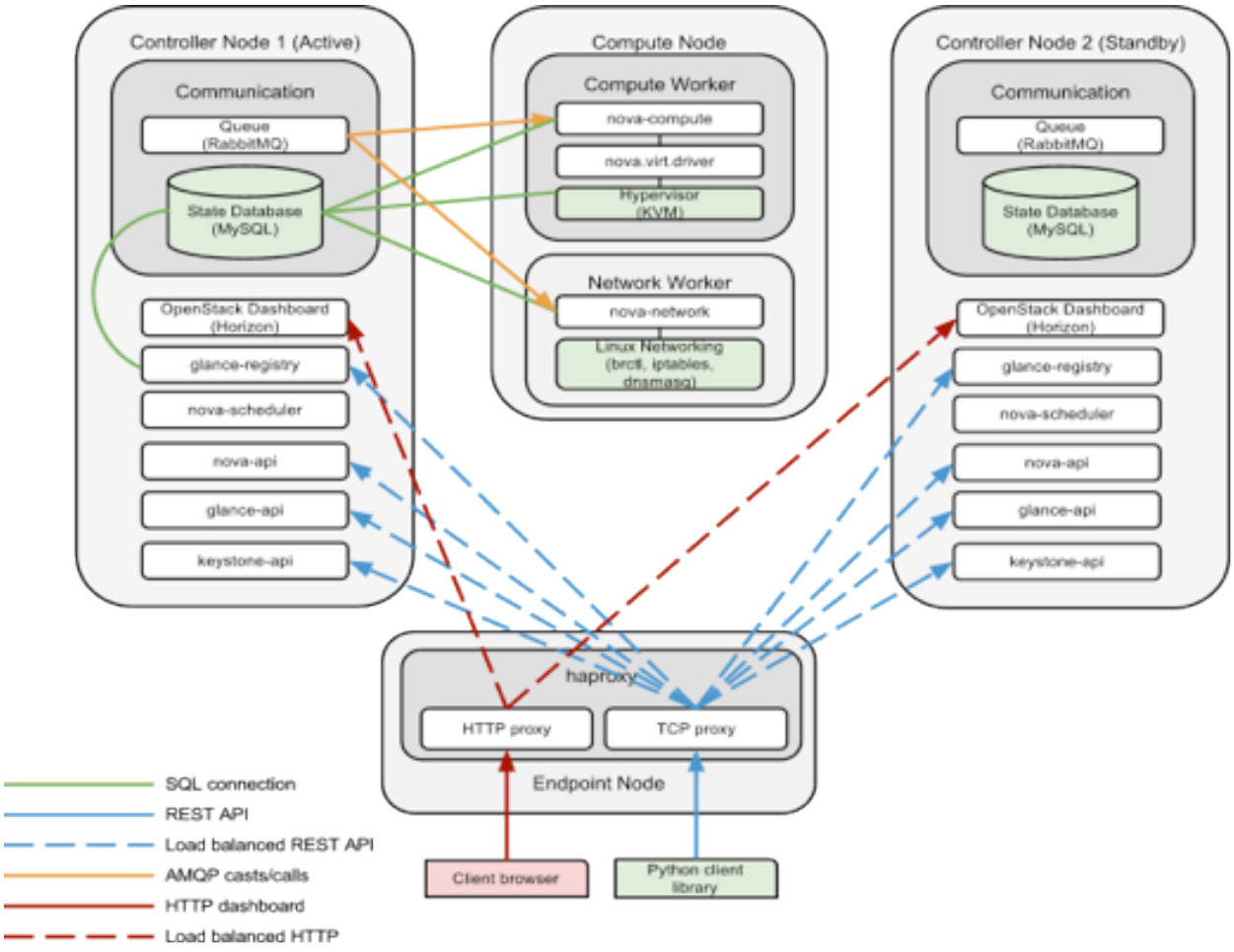

Figure 5.10: Openstack Compute nodes High Availability topology

Part I

Regulations

Cloud computing technologies are being implemented in a range of architectures, hardware, and primarily exist to facilitate other software services to run on top of them.

Because of its very nature, regulations around the cloud cannot be enforced very heavily, but instead most of the regulation pressure is on the clients that actually deploy code under regulations on the cloud. Nonetheless, this annex will contain the regulations cloud providers in the EU should keep in mind, most of them related with the topic of privacy, security, and retention of data.

Application of the law Current law makes a distinction between *data controllers* and *data processors.* The assumption is that data processors, such as the tools provided by the cloud, are performing arbitrary code and cannot be liable. On the other hand, data controllers are liable and responsible of the law violations that may happen.

National law will only apply when:

- establishment of EU-based controller located in its territory processes personal data
- controller outside EU uses equipment within territory

Transfer of data outside borders As of now, personal data should only be transferred across countries authorized by the EU. This includes all the state members, plus Switzerland, Argentina and Canada.

Of course, this would not work very well in practice as users might want their data from outside the EU borders. Exceptions to this law can happen if the user accepts a Service Level Agreement that permits data transfer across borders. In the case of the United States of America, a treaty between the EU and the USA allow this transfer of data to so-called "*safe harbours*", data centers who are trusted and follow the EU conventions.

EU General law The following laws are applicable under all jurisdictions in the EU. Member states cannot override in any way these directives. It is not a requirement by law, but it is common practice for data centers to ask for an audit of ISO/IEC 27001:2005, as it has set precedent case law at the European Court of Justice in Luxembourg.

- Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data
  - Personal data retention obligations
  - Tax related storage requirements
  - Labour law related storage requirements

- Directive 2000/31/EC of the European Parliament and of the Council of 8 June 2000 on certain legal aspects of information society services, in particular electronic commerce, in the Internal Market ('Directive on electronic commerce')
  - Protects host owners from being liable in case of an user storing illegal content in their servers, if and only if the host owner is unaware of this and takes measures if it happens.
  - Processing activities remain unprotected. For instance, it is not clear yet who is liable if a cloud provider is used to run a credit card fraud business.

Part II

Budget

The following budget takes into account the design explained in chapter 'Deployment'. Some recommendations from the Openstack Foundation will be used in order to determine which hardware will be necessary, and what will be the costs during a long period of time.

- Cloud Controller node (runs network, volume, API, scheduler and image services)
  - Any 64-bit x86
  - 12 GB RAM
  - 30 GB hard drive at least
  - 2 TB SATA disks for volume storage
  - 1 GB NIC
    - Suggested: HP DL360P
- Compute nodes (runs virtual instances)
  - Any 64-bit x86
  - 16 GB RAM
  - 30 GB hard drive at least
  - Two 1 GB NICs
    - Suggested: HP DL380P
- Database Nodes
  - 8 processor cores or greater
  - 16GB RAM or greater (serving +4000 nodes efficiently)
  - 2 TB
  - Requires InnoDB tuning to take advantage of these number of cores.
- HAProxy load balancing nodes
  - Two cores (HAProxy does not make use of more than that) 2.66 GHz - 40000 connnections/second
  - Can be virtualized
  - 16 GB RAM
  - Disk is irrelevant
  - At least 1GB NIC, 2 cards if possible

Other parts of the deployment can be virtualized given this infrastructure. Industry suggests every compute node is on average able to create 10 virtual machines.

Buy vs lease

Assuming a large sized cloud of 2000 machines, 200 HP DL380P would be needed for the compute nodes - at 1962.00 EUR each - total is nearly ~400000.00 EUR for only the compute nodes. Around 5 controllers could be needed, at a similar price that is 10000.00 EUR. MySQL servers, can run for nearly 6000 EUR, and using the MySQL Cluster model explained in chapter Deployment, management nodes can be virtualized. Using 6 MySQL nodes, that would be 36000.00 EUR. A total of 536000.00 EUR would be spent on hardware, making the VM cost around 236 EUR (forever).

However, the bulk of this money is spent on compute nodes. Since business normally do not know what their needs are from the beginning, this kind of hardware can be rent for 50.00 EUR a month. This lowers the effective cost of each VM, hardware-wise, to around 5 EUR a month per VM. It would take around 47 months, almost 4 years until the point buying the hardware would be amortized. This allows business to check if their approach to private clouds would financially make sense, and in the event of bankruptcy before 4 years, the company would save money.

Labor and phases

Most of the inspiration for this cloud builds upon the work I have done at the Agile Infrastructure team at CERN. Therefore, the rates will be according to rates at CERN, which in turn are similar to industry rates in technology hubs such as the Silicon Valley or New York City. A standard rate of 30 EUR per hour, for a full-time employee, can be considered.

The phases are not sequential as the work shifted depending on the needs at any particular moment. For that very reason, I did not include a monthly planning.

The work can be divided on several parts, counting from February 2013 until September 2013, around 32 weeks. Prices will be quoted per week, but in practice some of these projects were happening in parallel and therefore did take longer than what they should have taken without interruptions.

- Tools evaluation and understanding of the current systems
  - 3 weeks - 3600 EUR
- Puppet HA deployment
  - 6 weeks - 7200 EUR
  - Five other people worked on this.
- Foreman HA deployment
  - 4 weeks - 4800 EUR
  - One other person worked on this.

- Foreman collaborations and integrations with other products
  - 10 weeks - 12000 EUR
- IPMI migration from old power control tools to Foreman
  - 5 weeks - 6000 EUR
  - One other person worked on this.
- Support and urgent bug fixes
  - 2 weeks - 2400 EUR
- Collaborations with industry leaders
  - 2 weeks - 2400 EUR
  - Visited teams at eBay, Paypal, Rackspace and Red Hat to share experiences and learn.
  - Invited to speak at Puppetconf 2013, spent several days learning with similar teams.
- Total = 38400 EUR

The total amount does only take into account the work I have worked on. In fact, some of the phases I mentioned could not be finished without the help of coworkers, these phases are Puppet HA, Foreman HA, and IPMI migration.

Part III

Bibliography

Bibliography

[1] *Apache mod proxy balancer 2.2 documentation* (Apache Foundation). https://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html.

[2] *Aws cloudformation user guide* (Amazon). http://docs.aws.amazon.com/AWSCloudFormation/

[3] *Chef briefing*. http://www.opscode.com/chef/.

[4] *Compute and image system requirements* (Openstack Foundation). http://docs.openstack.org/folsom/openstack-compute/admin/content/compute-system-

[5] *Deployment topologies for ha with openstack cloud* (Mirantis). http://www.mirantis.com/blog/117072/.

[6] *Directive 2000/31/ec of the european parliament and of the council of 8 june 2000 on certain legal aspects of information society services, in particular electronic commerce, in the internal market ('directive on electronic commerce')* (EurLex). http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32000L0031:En:HTML.

[7] *Directive 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data* (EurLex). http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML.

[8] *Foreman manual 1.2* (Red Hat). http://theforeman.org/manuals/1.2/.

[9] *Haproxy 1.4 user guide* (Willy Tarreau). http://cbonte.github.io/haproxy-dconv/.

[10] *Hp proliant dl360p gen8 server* (HP). http://www8.hp.com/us/en/products/proliant-server

[11] *Hp proliant dl380p gen8 server* (HP). http://www8.hp.com/us/en/products/proliant-server

[12] *Iso/iec 27001:2005* (International Standards Organization). http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=

[13] *Mirantis ha queues and sql deployments* (Piotr Siwczak). http://www.mirantis.com/blog/ha-platform-components-mysql-rabbitmq/.

[14] *Mirantis understanding request flows - nova architecture* (Mirantis Team). http://www.mirantis.com/blog/117072/.

[15] *Openstack guide block storage guidelines* (Openstack Foundation). http://docs.openstack.org/grizzly/openstack-block-storage/admin/content/block_st

[16] *Openstack heat* (Openstack Foundation). https://wiki.openstack.org/wiki/Heat.

[17] *Openstack networking* (Openstack Foundation). https://wiki.openstack.org/wiki/Neutron.

[18] *Puppet documentation* (Puppetlabs). http://docs.puppetlabs.com/.

[19] *Rackspace openstack cloud on a budget* (Silva Tech Solutions). http://www.silvatechsolutions.com/tech-news/rackspace-openstack-cloud-on-a-budge

[20] *Service redundancy and traffic balancing using anycast* (Sean Jain Ellis). http://www.slideshare.net/bandarji/service-redundancy-and-traffic-balancing-usin

[21] *Sp800-144 guidelines on security and privacy in public cloud computing* (EurLex). http://csrc.nist.gov/publications/nistpubs/800-144/SP800-144.pdf.

[22] *Subcollectives reference* (Puppetlabs). http://docs.puppetlabs.com/mcollective/reference/

[23] *Swiftstack architecture* (SwiftStack). http://swiftstack.com/openstack-swift/architecture

[24] T. Bell, P. Andrade, J. Van Eldik, G. McCance, B. Panzer-Steindel, M. Coelho dos Santos, S. M. Traylen, and U. Schwickerath, *Review of CERN data centre infrastructure* (CHEP, Geneva, Dec. 2012). http://cds.cern.ch/record/1457989/files/chep%202012%20CERN%20infrastructure%20fi

[25] A. A. H. D. Chen Zhang, Hans De Sterck and R. Sladek, *Case study of scientic data processing on a cloud using hadoop* (Genome Quebec, McGill, University of Waterloo, May 2010). https://cs.uwaterloo.ca/~ashraf/pubs/hpcs09scientific.pdf.

[26] G. Lewis, *The role of standards in cloud-computing interoperability* (Carnegie Mellon University, Sep. 2012). http://www.sei.cmu.edu/reports/12tn012.pdf.

[27] P. Mell and T. Grance, *The NIST definition of cloud computing* (NIST, Sep.). http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[28] J. A. R. A. N. D. R. G. E. L. E. L. A. A. M. M. A. M. D. M. M. N. T. P. S. P. A. R. R. J.-L. R. B. P. R. D. R. M. B. S. K. S. P. S. R. S. D. T. K. V. Salvatore D'Agostino, Miha Ahronovitz and M. Versace, *Moving to the cloud* (NIST, Feb. 2011). http://cloudusecases.org/Moving_to_the_Cloud.pdf.